Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Informatica

Learning for scheduling a portfolio of constraint solvers

Algorithms and data structures

Relatore:
Chiar.mo Prof.
Zeynep Kiziltan

Correlatore:
Chiar.mo. Prof.
Barry O'Sullivan

Presentata da:
Luca Mandrioli

Sessione terza
2009/2010

Contents

Sommario

Nel lavoro di tesi qui presentato si indaga l'applicazione di tecniche di apprendimento mirate ad una più efficiente esecuzione di un portfolio di risolutore di vincoli (constraint solver). Un constraint solver è un programma che dato in input un problema di vincoli, elabora una soluzione mediante l'utilizzo di svariate tecniche. I problemi di vincoli sono altamente presenti nella vita reale. Esempi come l'organizzazione dei viaggi dei treni oppure la programmazione degli equipaggi di una compagnia aerea, sono tutti problemi di vincoli.

Un problema di vincoli è formalizzato da un problema di soddisfacimento di vincoli (CSP). Un CSP è descritto da un insieme di variabili che possono assumere valori appartenenti ad uno specifico dominio ed un insieme di vincoli che mettono in relazione variabili e valori assumibili da esse. Una tecnica per ottimizzare la risoluzione di tali problemi è quella suggerita da un approccio a portfolio. Tale tecnica, usata anche in ambiti come quelli economici, prevede la combinazione di più solver i quali assieme possono generare risultati migliori di un approccio a singolo solver.

In questo lavoro ci preoccupiamo di creare una nuova tecnica che combina un portfolio di constraint solver con tecniche di machine learning. Il machine learning è un campo di intelligenza artificiale che si pone l'obiettivo di immettere nelle macchine una sorta di 'intelligenza'. Un esempio applicativo potrebbe essere quello di valutare i casi passati di un problema ed usarli in futuro per fare scelte. Tale processo è riscontrato anche a livello cognitivo umano. Nello specifico, vogliamo ragionare in termini di classificazione. Una classificazione corrisponde ad assegnare ad un insieme di caratteristiche in input, un valore discreto in output, come *vero* o *falso* se una mail è classificata come spam o meno. La fase di apprendimento sarà svolta utilizzando una parte di CPHydra, un portfolio di constraint solver sviluppato presso la University College of Cork (UCC). Di tale algoritmo a portfolio verranno utilizzate solamente le caratteristiche usate per descrivere determinati aspetti di un CSP rispetto ad un altro; queste caratteristiche vengono altresì dette features.

Creeremo quindi una serie di classificatori basati sullo specifico comportamento dei solver.

La combinazione di tali classificatori con l'approccio a portfolio sarà finalizzata allo scopo di valutare che le feature di CPHydra siano buone e che i classificatori basati su tali feature siano affidabili. Per giustificare il primo risultato, effettueremo un confronto con uno dei migliori portfolio allo stato dell'arte, SATzilla.

Una volta stabilita la bontà delle features utilizzate per le classificazioni, andremo a risolvere i problemi simulando uno scheduler. Tali simulazioni testeranno diverse regole costruite con classificatori precedentemente introdotti. Prima agiremo su uno scenario ad un processore e successivamente ci espanderemo ad uno scenario multi processore. In questi esperimenti andremo a verificare che, le prestazioni ottenute tramite l'applicazione delle regole create appositamente sui classificatori, abbiano risultati migliori rispetto ad un'esecuzione limitata all'utilizzo del migliore solver del portfolio.

I lavoro di tesi è stato svolto in collaborazione con il centro di ricerca 4C presso University College Cork. Su questo lavoro è stato elaborato e sottomesso un articolo scientifico alla International Joint Conference of Artificial Intelligence (IJCAI) 2011. Al momento della consegna della tesi non siamo ancora stati informati dell'accettazione di tale articolo. Comunque, le risposte dei revisori hanno indicato che tale metodo presentato risulta interessante.

Chapter 1

Introduction

1.1 Background

In a world where we always aim to be faster, from building better mechanics/electronics parts in order to win races, to increase the CPU performance allowing us to run applications that we could only dream about years ago, most research in computing tries to push the throttle for increasing performances or decreasing the time requested for solving a specific task.

The challenge is also found in the world of constraint problems. In real life, many problems are treated as constraint problems, from train scheduling to crew scheduling of a flight company. There are also simpler examples that touch our everyday life, like the crosswords or sudoku games. Constraint problems can be formalised as constraint satisfaction problems (CSPs). A CSP is defined by a finite set of variables where each one can be associated to a domain of values. A set of constraints defines then the possible assignments of values to the variables [24].

The research in this field has been interested for quite many years, developing knowledge and new techniques. This field that worries about the solving paradigm of CSPs is called constraint programming [34]. The idea of constraint programming is that a problem is stated as a CSP and a general purpose constraint solver solves it. A constraint solver implements a series of techniques able to find a solution of the problem submitted, possibly in an optimal way. Nowadays many solvers exist and some of them participate to international competition with the purpose to elect the best constraint solver at the state of arts.

Apart from the development of new and optimised constraint solvers, new ideas have been developed for obtaining solutions of constraint satisfaction problems more efficiently. One of these is the portfolio approach [12]. This approach, also used in economical envi-

ronments, consists of the idea of using a set of solvers. These, combined together, try to optimize certain parameters, like minimising the solving time or maximising the number of instances solved in a determined amount of time.

The portfolio approach has been already used in different context, like for Satisfiability problems (SAT) problems [43]. SAT is a specialization of CSP, with a domain of values restricted to *true* and *false* and the constraints are Boolean formulas expressed in conjunctive normal form (conjunctions of disjunctions of literals). The previous citation is an example of integration of portfolio approach with machine learning techniques. Machine learning [27] is a field of artificial intelligence that employs a series of techniques to extract knowledge, in general, from past cases and use it in future. This general approach is also well known in field like cognitive science. In the portfolio case a machine learning algorithm could be useful deciding which solver to run looking on similarities of problems treated in the past.

1.2 Motivation and goals

In this work we take the challenge of creating a new way of combining a portfolio approach with machine learning techniques. At the state of art the closest examples that can be found are CPHydra and SATzilla. CPHydra [30] is a portfolio for solving CSPs developed at the 4C research centre in University College of Cork (UCC).[1] It combines a portfolio approach with case based reasoning (CBR), a popular machine learning technique. Using similarities in past cases and partitioning CPU-Time between the components of the portfolio, CPHydra maximises the expected number of solved problem instances within a fixed time limit. The previously mentioned SATzilla [43], is a portfolio for SAT solving combined with machine learning technique which forecasts the expected solving time of a given problem.

In this thesis, we continue in this successful line of research which exploits MLA in the construction of a portfolio of CSP solver. The originality of our work is that we want to explore a different way of exploiting machine learning algorithms in a portfolio. In the specific, the goal of the new approach presented is to show that exploiting classifications on CSPs is an efficient and successful strategy able to find solutions to CSPs efficiently. A classification is a methodology that, given a series of attributes in input, is giving a prediction expressed with a number of discrete categories in output, like *yes/no* to classify an email spam or not spam, or *easy*, *medium*, *hard* to classify the difficulty of a problem. Our work is using the CPHydra attributes. These attributes used are called also *features* and are characteristics that describe a CSP. These features are useful to

[1] http://4c.ucc.ie/web/index.jsp

machine learning algorithms to depict important information about the constraint problem analysed. Using these features we want to create reliable and competitive classifiers.

To test that classification applied to a portfolio approach is a successful technique, we want to achieve better performance compared to a configuration employing the best single solver of the portfolio. In doing this, we want to assure that this supremacy holds both in a single processor and in a multiple processors scenario. In the single processor case we want to test the case base where our approach has to perform well otherwise any further expansion would be useless. While, in the multiple processors case we want to test that our method is scalable through a multiple CPUs scenario.

1.3 Overview

In the hereby presented work we will first focus on establishing whether CPHydra features are good. In order to do that, we will first create a group of classifiers extracted from analysis on the run time distribution of the portfolio solvers. On these classifiers, we will compare CPHydra features, after an optimization work, to the features of one of the finest portfolio solver, SATzilla[43]. We will also verify that the classifiers created are reliable by means of determined statistics.

Once the quality of the features and the reliability of the classifiers are established, in our next phase we assembles the information extracted from classifiers. This information will be used for optimising a specific statistical measure obtained solving CSPs. In our work we want to minimise the average finishing time, the time by which an instance is solved. Minimising the average finishing time can be achieved by solving each instance with an increasing order of difficulty. In the portfolio context doing this corresponds to ordering the instances accordingly to their degree of hardness and then employing the fastest solver to solve the instance. Such ordering of the instances will be consistently smart to improve the usage of the portfolio. In doing that, we want to be sure that we outperform the performances of a system composed by the best constraint solver of the portfolio: *mistral* [16]. This has to be verified both in a single processor case and a multiple processor case for establishing that our solution scales correctly in a parallel setting.

The work here presented is structured in the following way: Chapter 2 will introduce the background concepts about machine learning, constraint satisfaction problems, constraint programming and portfolio approach. Chapter 3 will go into the learning part, talking about the classifiers built, features selection, and features set comparison. Chapter 4 will consider the second main part of the work, where we will build scheduling rules based on classifiers. First we will simulate the performance of such scheduling rules in a single processor case and, in a second moment, in a parallel static context. Finally,

in Chapter 5 we will survey some related works before concluding and discussing future works in Chapter 6.

The hereby presented work resulted to a paper submission to the 2011 International Joint Conference of Artificial Intelligence (IJCAI).[2] By the time of submitting the thesis we are not yet informed about its acceptance status. However the reviewers feedback has revealed that our method is appealing. The submitted paper is a result of a collaboration with the 4C research centre in University College of Cork.

[2]http://ijcai-11.iiia.csic.es/

Chapter 2

Background

In this chapter, the required background on the thesis work is provided. However, due to the vastness of each topic treated, this chapter will be focus only on the aspects which are necessary for understanding of the presented work. The background chapter consists of an introduction to machine learning and its algorithms; a section reserved to introduce constraint satisfaction problems and constraint programming paradigms and a section introducing the principles of algorithm portfolio.

2.1 Machine learning

Since the creation of computers, there was always interest in making the machines learn and make them able to acquire knowledge from the environment. Starting from the sci-fi film category, through the Isaac Asimov literature, many aspects about robotics and machine learning were covered, mainly for entertainment purpose (robots turning against the human beings). The reality is far away from the Hollywood point of view.

Nowadays, the fields in which learning techniques are applied are many and not related only to computer science. In the following we present some examples.

Games: there are several papers related to the topic of making a machine learn how to play chess against human and competing on the better intelligence. Since then, all the video games started to develop better learning agents/players. Such an example can be found in a particular field called general game playing. In [8] an agent is developed with the aim of create an intelligence that can automatically and in real-time learn how to play many different games at an expert level without any human intervention.

Patter recognition: from recognizing handwritten digit/text, to speech recognitions are all subjects that requires learning [15].

Robotics: many are the aspects where learning in robotics is important. Depending on the purpose of the robot, there could be learning algorithm regarding interaction skills, working procedures. The classic example is the vacuum cleaner that analyses its previous behaviour for taking further decision, like clean first a particular spot which is usually dirty or either learn over obstacle objects [36].

Biology and medicine: there are numerous cases [28] [21] in which learning is required in these fields, for example predicting various form of cancer like prostate cancer or women breast cancer [2] and for measuring the DNA microarrays expression [15]. These DNA microarrays evaluate the eventual presence of a specific gene in a given cell and, if affirmative, the presence of a determined molecule in it .

Business: interesting applications are the predictions of important parameters like risk factors for loans or bankruptcy [26]. Among the others there are specific works on agent learning, for example, on oligopolistic competition in electricity auctions [13].

This section continues with an introduction on the main type of learning, a distinction between lazy and eager learning, as well as a list of the main machine learning algorithms and performances metrics used to evaluate the results produced.

2.1.1 Supervised, unsupervised and reinforcement learning

Before taking a step further, let us give a formal definition of learning according to Mitchell's "Machine Learning" [27]:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Let us take a concrete example. Imagine a software able to recognize handwritten digits for the ZIP code. The *task* T, that the software has to accomplish, is to recognize the handwritten digits from the image provided. The so called *experience* E is a database of images containing digits and respectively the output result from which the software will analyse similarities, patterns and learn. The *performance* P measure is given by the percentage of images correctly recognized by the software.

This is a classic example of a *supervised learning* problem. Such a problem is easily identifiable by a set of variables in input, measured or given, that are having influences on the output(s). The word supervised is referred to the goal which is to use inputs to

predict the output values. This type of learning is often referred as "learning with a teacher" [15]. Under this metaphor the "student" presents an answer $\hat{y}$ for each x_i in the training sample, and the supervisor or "teacher" provides either the correct answer y_i and/or an error associated with the student's answer. This error is usually characterized by some loss function $L(y, \hat{y})$, for example, $L(y, \hat{y}) = (y - \hat{y})^2$.

Examples such as handwritten digit recognition or spam recognition in emails, in which the aim is to assign a finite number of discrete categories (i.e. given an email classify it spam or not spam), are called *classification problems*. If the searched output consists of one or more continuous variables, then the task is called *regression*. A discrete value can assume only a finite set of values, while a continuous can assume infinite different values. An example of a regression problem is the prediction of the prostate-specific antigen (PSA) value. Such a continuous value for example, measures the proteins produced by cells of the prostate gland in the blood. The higher this values is, the more likely it is that this specific cancer is present.

With a bit more of formality, it can be said that, given a vector of input data X, where $X = x_0, ..., x_n$, a machine learning algorithm (MLA) is a function f that returns a so called prediction $\hat{\mathrm{Y}}$.

$$f(X) = \hat{Y}$$

If the output is discrete, the function f is called *classifier*, otherwise *regression*, if the output is continuous.

In other pattern recognition problems, the training data consists of a set of input vectors without any corresponding target values. Such problems are called *unsupervised learning problems* [15]. Their goal may be to discover groups of similar examples within the data (*clustering*), or to determine the distribution of data within the input space (*density estimation*). This type of learning is done without the help of a supervisor or teacher providing correct answers or a degree of error for each observation, that is why is also known as "learning without a teacher". The previously introduced DNA microarrays expression is an example of unsupervised learning problem, since a DNA microarrays expression does not need to be in a class, but it needs to be clustered or analysed in other expressions so that patterns might be found.

The last of the most famous learning type, which is used in applications like the aforementioned robotics, is the *reinforcement learning*. This technique [3] is interested in the problem of finding suitable actions to take in a given situation in order to maximize a reward. Those actions are discovered by a process of trial and error which characterizes the reinforcement learning compared to the more classic supervised one. In fact, the

learning process comes from examples provided by some knowledgeable external supervisor that alone is not adequate for learning from interaction [39].

These activities can be easily summarised in the following statements. A reinforcement learning agent and its environment interact over a sequence of discrete time steps. The specification of their interface defines a particular task: the actions made by the agents are the choices, the states are basis for making the choices and the rewards are the basis for evaluating the choices made. Everything inside the agent is completely known and controllable by the agent while everything outside is incompletely controllable but may or may not be completely known. Given a stochastic rule by which the agent selects actions as a function of states (a policy), the agent's goal is then to maximize the amount of reward it receives over time and choices made.

There is one feature that is worth to be mentioned about reinforcement learning which regards exploration and exploitation. It is very important, in fact, to monitor the trade-off between exploration, in which the system tries out new kinds of actions to see how effective they are, and exploitation, in which the system makes use of actions that are known to yield a high reward. It has to be a balanced choice between the two because being too strongly focused on either exploration or exploitation will bring poor results [39].

2.1.2 Lazy and eager learning

Lazy methods are called in such way because they defer the decision of how to generalize beyond the training data until each new query instance is encountered. On the other hand, methods that are called *eager* generalize beyond the training data making possible all the operations that define the approximation of the algorithm to the target function before observing the new query [27].

Let us distinguish two differences between lazy and eager learning:

- in computation time;
- in the classifications produced for new queries.

In the first case, lazy methods will generally require less computation time during training, but more computation time when they have to predict the target value for a new query. In the second, lazy methods may consider the query instance x_q, when deciding how to generalize beyond the training data. While, in the eager methods, by the time they observe x_q, they have already chosen their (global) approximation to the target function. In other words, a lazy algorithm might take decision at the querying time, while an eager algorithm cannot.

2.1.3 Machine learning algorithms

The list of machine learning algorithms (MLA) implemented so far is very long and some of them require advanced mathematical/statistical background to be completely clear. The purpose of this section, is to give an overview of the most important MLA operating mainly in supervised learning, since the thesis work is interested on these kind of learning problems. For the sake of simplicity the algorithms are grouped in the closest manner possible to the categorization method of Weka [17], a data mining tool used for this project.[1]

The principal categories are the following:

- artificial neural networks;
- Bayesian learning methods;
- decision trees;
- lazy learners;
- meta classifiers;
- regression;
- rules;
- support vector machine.

Artificial Neural Network (ANN) [27] is a method that tries to resemble to the human neural network. The most famous type of ANN, multi layer perceptron, is based on the perceptrons, the artificial equivalent of the neurons. Artificial neural networks are built out of a densely interconnected set of perceptrons, where each one of them takes a number of real valued inputs (possibly the outputs of other units) and produces a single real valued output (which may become the input to other units). These inputs and outputs influence the interconnected perceptrons making the neural network adaptive to the training inputs. ANN can be a powerful tool but is usually complex to find a good tuning of the parameter and also they require a long training time.

In the *Bayesian* [27] learning methods, the algorithms are based on the Bayes formula, which determines the most likely hypothesis from some space H, given an observed

[1] `http://www.cs.waikato.ac.nz/ml/weka/`

training data set D. The Bayes formula specifies a way to calculate the probability of an hypothesis h in the following way:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

where $P(h)$, called the prior probability of h, measures the probability of the correctness of the hypothesis h, before having seen the training data. $P(D)$ denotes the probability that training data D will be observed. $P(D|h)$ denotes the probability of observing data D knowing that the hypothesis h is valid or holds. $P(h|D)$ is the posterior probability of h and it reflects a level of confidence on h holding after the training data D is seen. A remark is that the posterior probability $P(h|D)$, in contrast to the prior probability $P(h)$, is influenced by the training data D. Among the variety of Bayesian learning methods one worth to be mentioned is the naive Bayes learner. In some domains its performance showed to be comparable to that of neural network and decision tree learning [25].

Decision trees algorithms, like the name says, are methods that construct a tree easily readable following the branching sequence that, from the root node to the leaf node, leads to a classification decision [27]. Each node of the tree specifies a test of some attribute of the instance, while each tree branch from a node, corresponds to one of the possible values of this attribute. Decision trees usually work with entropy evaluation once it comes to select which attribute branch first and are the most direct example of eager learning type. An example of classic decision trees is *C 4.5* [33].

Decision trees are nowadays brought to a powerful form of classification called *Random Forest* [4]. This algorithm uses the technique known as *bagging*. This method works by training each algorithm employed, in this case random trees, on a sample chosen randomly from the original training set. The multiple output computed are then combined using simple voting. The final composed output classifies an example x to the class most often assigned by the underlying multiple output previously computed. In general, decision trees are algorithms that suit better for classification problems [27].

In the *lazy learning* category, discussed above, the most important algorithm is *k-NN*. The k-nearest neighbor [15] is a lazy learning algorithm that returns the k most similar cases to the new problem submitted, where k is a parameter of the algorithm. To define how to evaluate the similarity between the cases, a metric is required, which normally is the Euclidean distance.

Case based reasoning (CBR) [1] is a lazy learning method that allows to elaborate solutions of new appearing problems, choosing from similar cases already encountered and solved. An important feature is that CBR is an approach to incremental, sustained learning, since a new experience is retained each time a problem has been solved, making it

immediately available for future problems. Generally the CBR activities are summarized by the following list:

1. **Retrieve** the most similar cases;
2. **Reuse** the information and knowledge in that case to solve the problem;
3. **Revise** the proposed solution;
4. **Retain** the parts of this experience likely to be useful for future problem solving.

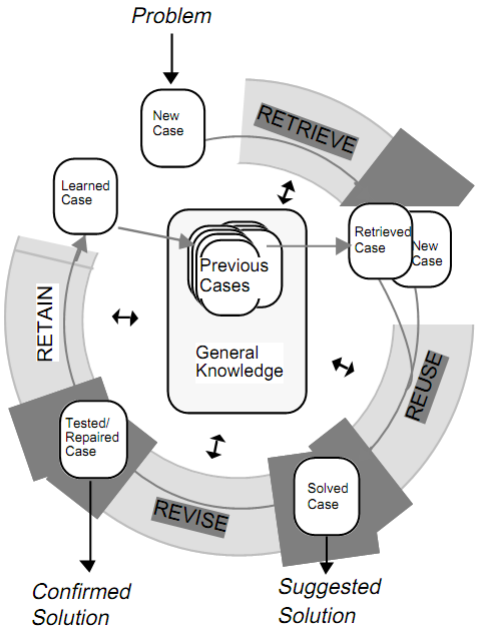


Figure 2.1: CBR: the cyclic process.

In Figure 2.1 the typical cyclic process of a CBR is depicted. The new problem is submitted to the CBR which, with the help of a k-nearest neighbor (k-NN) algorithm, retrieves the similar cases. Such similar cases combined sometimes with general knowledge are being reused for producing a solution of the problem submitted. Through the revise process the solution is then validated and corrected if the process is not successful. The last phase of retaining consists in extracting the useful information able to increase the cases database for the future usage.

A CBR algorithm looks like an implementation of a reinforcement learning method.

This is not completely true. Even if there are similarities in these approaches, there is a substantial difference in the way they are learning. CBR uses the knowledge of the past cases, or in some cases the general world knowledge, while reinforcement learning techniques acquire knowledge exclusively by means of a process of trials and errors.

In the *meta classifiers* family there are all those algorithms powered by an inner MLA combined with techniques like bagging, boosting or wagging. Some examples of meta classifiers are *AdaBoost*, *MultiBoost* and *Random Committee*. To describe the main idea of these classifiers, let us analyse briefly *AdaBoost*. *AdaBoost* [9] is based on the technique of *boosting*. Such a method works by repeatedly running a given weak learning algorithm on various distributions over the training data, and combining the classifiers produced by the weak learner into a single classifier. More specifically, *AdaBoost* calls the weak learn algorithm (WL) repeatedly in a series of rounds. On each round, *AdaBoost* provides WL with a distribution over the training set. In response, WL computes a classifier or hypothesis which should correctly classify a fraction of the training set that has large probability with respect to. The weak learner's goal is to find an hypothesis which minimises the training error. At the end of the rounds *AdaBoost* combines the weak hypotheses into a single final one. The difference between *AdaBoost* (mainly the boosting methods) from a bagging technique is that boosting is iterative. Whereas in bagging individual models are built separately, in boosting each new model is influenced by the performance of those built previously. Another important advantage coming from the iterative nature is that boosting allows weighting a model's contribution by its performance rather than giving equal weight to all models, like in bagging.

MultiBoost [40] is actually an extension of *AdaBoost* that combines *AdaBoost* with wagging. *Wagging* is variant of bagging, which requires a base learning algorithm that can utilize training cases with differing weights. Rather than using random samples to form the successive training sets, wagging assigns random weights to the cases in each training set. *Random Committee* [41] builds an ensemble of weak classifiers and averages their predictions. *Random Committee* distinguishes itself by making each classifier be based on the same data but using a different random number seed. This only makes sense if the base classifier is randomized, otherwise the classifiers would all be the same.

The simplest algorithm of regression category is the *linear regression* [27]. Such an algorithm approximates a linear function f to a given set of training samples close to the query instance x.

$$f(x) = w_0 + w_1 a_1(x) + w_2 a_2(x) + \ldots + w_k a_k(x)$$

In the specific, $a_i(x)$ indicates the ith attribute value of the instance x, while w_i are the coefficients weighting each attribute. This is a natural technique considered when the class and all the attributes are continuous values.

In the *rules* category are grouped all those classifiers based on simple rules [18], like 1-R, a classifier that learns one rule from the training set.

A *Support Vector Machine* (SVM) is a method that aims to find the optimal separating hyperplane that separates the instances of a two classes problem (separable or not). In order to find this optimal solution, SVM uses a margin, defined as the smallest distance between the decision boundary and any instance of the training set. SVM then, separates the two classes' problem, maximising the margin between the training points and the decision boundary.

SVM is not the only algorithm that is separating the classes. For example the multi layer perceptron algorithm, introduced under the ANN category, tries to find a separating hyperplane minimizing the distance of misclassified points to the decision boundary. However this solution is dependent on the initial parameter of the perceptrons and the solution is found just one of the many available. SVM has different variants, like the one that can be applied to regression problems or for example the *MultiClass* SVM that allows the algorithm to manage classification with multiple classes since the classic SVM is meant to be for two classes problems only [15].

2.1.4 Performances metrics and testing techniques in classification

There are many metrics able to depict the goodness of a classification in the supervised learning context, but without a doubt the first one that has to be considered is the accuracy. The *accuracy* of a machine learning algorithm represents, in percentage, how many instances are correctly classified.

Another metric, Cohen's kappa statistic (κ-*statistic*) [5] is used to measure the agreement between the predicted and the observed classification. It is expressed with a value from 0 to 1, where values closer to 0 represent a poor agreement, while values closer to 1, a good agreement. However, this measure does not take misclassification costs into account.

Now we will show a practical example using an outcome from an experiment like the one used in [14]. In Table 2.1 two type of classifiers (A,B) classify n instances in two classes $+$ and $-$. a and d represent the number of instances classified respectively + and - from both classifiers A and B. b and c, instead, represent the number of instances which are classified + by one classifier and - from the other. A_+, A_-, B_+ and B_- are the sum of all the instances classified respectively + and -, by A and by B individually.

Classifier B	Classifier A		Total
	+	-	
+	a	b	B_+
-	c	d	B_-
Total	A_+	A_-	n

Table 2.1: Distribution of n instances classified by two classifiers.

The formula of Cohen's κ-statistic is the following:

$$\kappa = \frac{P(a) - P(e)}{1 - P(e)}$$

where the probability of agreement (meaning that both classifiers have the same output) $P(a)$, is defined as:

$$P(a) = \frac{a + d}{n}$$

the probability of expected agreement $P(e)$ is defined as follows:

$$P(e) = P(A_+)P(B_+) + P(A_-)P(B_-)$$

and

$$P(A_+) = \frac{A_+}{n}, P(B_+) = \frac{B_+}{n}, P(A_-) = \frac{A_-}{n}, P(B_-) = \frac{B_-}{n}$$

In the Tables 2.2 and 2.3 two examples that show how κ is varying through different data are given. Let us take the following tables with both $n = 100$ instances over two classes and two classifiers (the problem can be extended to multiple classes):

Classifier B	Classifier A		Total
	+	-	
+	45	15	60
-	25	15	40
Total	70	30	100

Table 2.2: κ statistic calculation example with 100 instances.

$$P(a) = \frac{45 + 15}{100} = 0.6$$

$$P(e) = 0.7 \times 0.6 + 0.3 \times 0.4 = 0.42 + 0.12 = 0.54$$

$$\kappa_1 = \frac{0.6 - 0.54}{1 - 0.54} = 0.1304$$

Classifier B	Classifier A		Total
	+	-	
+	25	5	30
-	5	65	70
Total	30	70	100

Table 2.3: κ statistic calculation example with 100 instances.

$$P(a) = \frac{25 + 65}{100} = 0.9$$
$$P(e) = 0.3 \times 0.3 + 0.7 \times 0.7 = 0.09 + 0.49 = 0.58$$
$$\kappa_2 = \frac{0.9 - 0.58}{1 - 0.58} = 0.7619$$

In Table 2.2, the level of disagreement is higher than the one in Table 2.3 and is testified by their κ values: $\kappa_2 = 0.7619$, $\kappa_1 = 0.1304$.

The most important testing technique in classification is the *n-fold cross validation* [27]. A cross validation is a methodology that consists of dividing the original data set in n-folds (typically 5 or 10). Iteratively then, one fold is kept as testing set while the remaining $n - 1$ folds are used for training the classifier. The cross validation is usually adopted whenever the data set is not big enough to allow two consistent sets, one for training and the other one for testing. Furthermore it might be useful to reduce the problem of overfitting which happens usually in the decision tree learning when there is noise in the data (wrong instances) or when the training set is too small for having a good representation of the problem. An algorithm is said to be *overfitting* when an increase of performance of the classifier on the training set, corresponds to a decrease on the test set. This results in taking the training data too much as a strict example of reality, risking that the algorithm is not able to later generalize on new presented problems. An evident drawback of cross-validation is that the number of training runs required are increasing as a factor n and this can be problematic for those models that are computationally expensive to train.

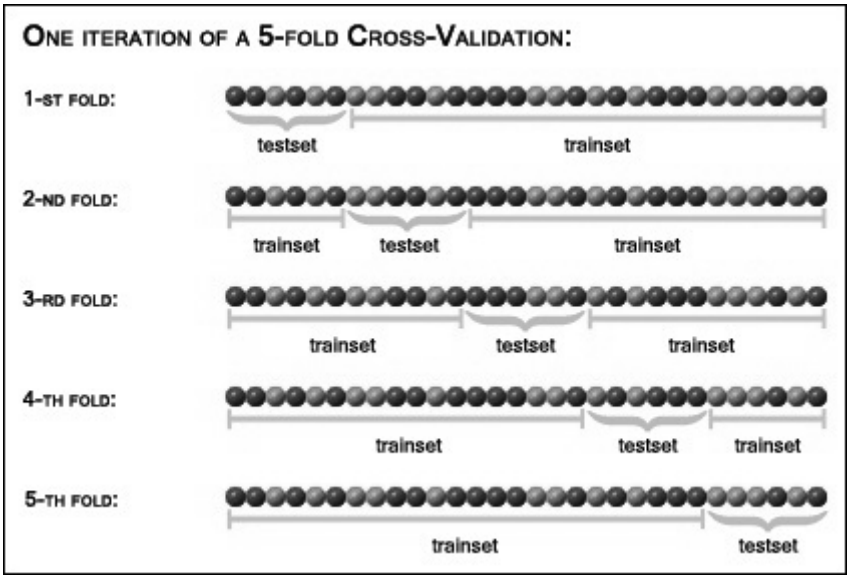


Figure 2.2: An example of a 5-fold cross validation.

The Figure 2.2 gives an example of 5-fold cross validation where each fold consists of six instances represented as dots.[2] As it is possible to see, in the first iteration only first fold is kept as test set, while the remaining ones are used as the training set. Once it iterates over the next fold, the fold is selected as new test set. This process continues until all the five folds are considered.

2.2 Constraint satisfaction problems and constraint programming principles

In this section we will focus on the basic principles of constraint satisfaction problems, giving an overview from the theoretical point of view, introducing also one of its special case, the satisfiability problems. Then, we will present some basic concepts of constraint programming and some principles and examples of constraint solvers.

2.2.1 Constraint satisfaction problems

The every day life is full of constraint problems. When it comes to scheduling trains or scheduling crews for a flight company, these are all problems that deal with constraints. Even in the everyday games, like crosswords and sudoku, we can find constraints problems. Given a real world constraint problem, it is necessary to define a mechanism that

[2]Image from `http://genome.tugraz.at/proclassify/help/pages/XV.html`

allows to formalize the problem in a schematic way.

Such problems can be formulated as *Constraint Satisfaction Problem* (CSP), defined in the *Handbook of constraint programming* [34] as:

> A CSP P is a triple $P = \langle X, D, C \rangle$ where:
>
> - X is an n-tuple of variables $X = \langle x_1, x_2, \ldots, x_n \rangle$;
> - D is a corresponding n-tuple of domains $D = \langle D_1, D_2, \ldots, D_n \rangle$ such that $x_i \in D_i$;
> - C is a t-tuple of constraints $C = \langle C_1, C_2, \ldots, C_t \rangle$.

A CSP is defined by a finite set of variables, each of which is associated with a domain of possible values that can be assigned to the variable, and a set of constraints that define the set of allowed assignments of values to the variables [24]. A *constraint* C_j is a pair $\langle R_{S_j}, S_j \rangle$ where R_{S_j} is a relation on the variables in S_j. In other words, each constraints C_j involves some subset of the variables S_j and specifies in R_{S_j} the set of allowed combinations of values that the variables can take simultaneously. In such way, R_{S_j} can be defined as a subset of the Cartesian product of the domains of the variables in S_j.

Given a CSP, the task is normally to find an assignment to the variables that satisfies the constraints, which we refer to as a solution. A *solution* P is an n-tuple $A = \langle a_1, a_2, \ldots, a_n \rangle$ where $a_i \in D_i$ and each C_j is satisfied in that R_{S_j} and it holds the projection of A onto the scope S_j .

Given a CSP, it is requested to find one of the following:

- any solution, if one exists;
- all the solutions;
- an optimal solution given some objective function defined in terms of apart or the entire set of variables.

A classic example used in literature: the n-queens problem. Such a problem consist of finding the right disposal point of n queens on a $n \times n$ chessboard, in a way that each queen is not able to attack and being attacked by another one (considering that a queen moves horizontally, vertically and diagonally).

One possible way to encode this problem as CSP is as follows:

- a variable x_i is defined for the queen on row i;
- the domains D_i of each variable x_i is defined as $D_i = \{1 \dots n\}$, giving the possible columns that queen i can be placed;
- $\forall i, j, i \neq j$ having $1 \leq i < j \leq 8$, the constraints are defined as:
  1. $x_i \neq x_j$ (knowing that two queens cannot be on the same row from the $i \neq j$, here we say that two queens cannot be placed on the same column);
  2. $i - j \neq x_i - x_j$ (no other queen can be placed on one of the two diagonals);
  3. $j - i \neq x_j - x_i$ (no other queen can be placed on other one of the two diagonals).

Figure 2.3 shows one of the possible solutions of the 8×8 queens problem. It is verifiable that no queen on the board is able to attack another one.

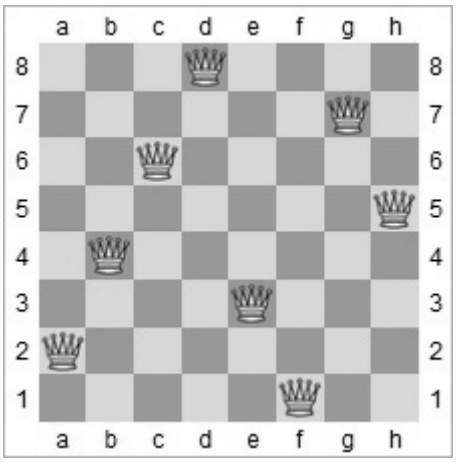
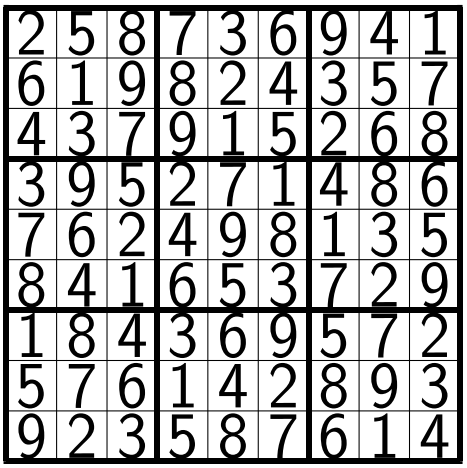


Figure 2.3: One possible solution of the 8-queens CSP.

Another example that helps to better understand what CSPs are, is the popular sudoku game. A classic sudoku problem [37] can be expressed in the natural language like the following: given a 9×9 board, the purpose of the game is to fill each of the 81 boxes with a number from 1 to 9 in a way that in each row, in each column and in each 9 major 3×3 subregion, is possible to have all the numbers from 1 to 9. An solution of a sudoku game looks like the following:



One possible way to encode this problem as CSP is as follows:

- each variable $x_{i,j}$ corresponds to the cell on the board, with $i, j \in [1, 9]$;
- the domain $D(x_{i,j}) = \{1 \dots 9\}$, giving the numbers that can be filled in each cell;
- and the constraints are :
  1. $\forall i, j_1, j_2 \in [1, 9]$, then $x_{i,j_1} \neq x_{i,j_2}$ (two numbers in the same row cannot be equal);
  2. $\forall j, i_1, i_2 \in [1, 9]$, then $x_{i_1,j} \neq x_{i_2,j}$ (two number in the same column cannot be equal);
  3. for each 9 subregions S_r, $x_{ij} \neq x_{hk}$, if $i, j, h, k \in S_r$, $i \neq h$ and $j \neq k$ (two numbers in the same major 3×3 subregion cannot be equal).

The *arity* of a constraint is the number of variables it constrains. This is an important feature that can be used for analysing and depicting the specificity and possibly the complexity of CSP. A first classification among CSP comes precisely on this feature. Constraints are defined as: *unary*, *binary* or *n-ary* if the arity is equal to 1, 2 or > 2 respectively. *Intensional constraints* are specified by a formula that is the characteristic function of the constraint. *Extensional constraints* are specified by a list of its satisfying tuples. *Global constraints* are those constraints that involve relations between an arbitrary numbers of variables. The most used example of a global constraint is the `all-different` constraint which states that the variables in the constraint must be pairwise different [34].

2.2.2 SAT

Satisfiability problems (SAT) are actually a specialization of CSP, where the domains are restricted to be $\{true, false\}$ and the constraints are clauses expressed as a Boolean formula in conjunctive normal form (CNF) [34].

An example of a SAT problem can be the following: the formula $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_3)$ is a clause expressed in CNF, conjunction of disjunction of literals, where a literal is a Boolean variable or its negation. A solution to this problem is: $x_1 = F; x_2 = F; x_3 = T$.

The satisfiability of propositional formulae is one of the most famous example of problems in computer science, mainly because of its interesting applications in the theoretical field. It has been proved that problem in the $\mathcal{NP}$ class [6] (problems for which exist an algorithms that given an instance i and one of its possible solution s, is possible to verify the correctness of s within a polynomial time) can be encoded into SAT ones and then be solved by SAT solvers.

2.2.3 Constraint programming

Constraint programming is a solving paradigm of constraint problems that draws on a wide range of techniques from artificial intelligence, computer science, databases, programming languages, and operations research. Constraint programming is currently applied with success to many domains like:

- scheduling;
- planning;
- vehicle routing (i.e. the travelling salesman problem (TSP): the problem of finding the shortest path that visits a set of customers and returns to the first one);
- configuration (the task of composing a customized system out of generic components. Classic examples involve constraints to assembling computers and cars);
- networks (apart from the classic networking field, CP can be used in electricity/water/oil networks);
- bioinformatics.

Constraints are just relations, and a constraint satisfaction problem (CSP) states which relations should hold among the given decision variables. For example, in scheduling activities in a company, the decision variables might be the starting times and the durations of the activities and the resources needed to perform them, and the constraints might be on the availability of the resources and on their use for a limited number of

activities at a time.

The basic idea in constraint programming is that the user states the constraints and a general purpose constraint solver is used to solve them. Constraint solvers take a real-world problem like the scheduling activities in a company previously mentioned, represented in terms of decision variables and constraints, and find an assignment to all the variables that satisfies the constraints.

2.2.4 Constraint solvers

Constraint solvers search the solution space using different techniques [34]. The main algorithmic techniques for solving CSPs are backtracking search, and local search. While the first one is an examples of *complete algorithm*, the local search is an example of *incomplete* algorithm. Complete means that there is the guarantee that a solution will be found, if one exists, otherwise prove unsatisfiability, or find an optimal solution. On the other hand incomplete algorithms are those that cannot assure to find a solution. Systematic methods, like the more advanced versions of *backtracking search*, interleave search and inference, which helps detect regions of the search space that does not contain solutions. This information is usually *propagated* among the constraints. *Constraint propagation* explicitly forbids values or combinations of values for some variables of a problem because a given subset of its constraints cannot be satisfied otherwise. For example, in a crossword puzzle, you propagate a constraint when the words *Iceland* and *Germany* are discarded from the set of European countries that can fit a 7-digit slot because the second letter must be a 'R'. The backtracking search is represented as a search tree where partial solutions are built up by choosing values for variables until a dead end is reached, where the partial solution could not be consistently extended. When the dead end is reached, the last choice is rolled back and another one tried (backtrack). This is done in a systematic manner that guarantees that all possibilities are tried.

A *local search* algorithm starts with an initial configuration (for the CSP are, typically a randomly chosen, complete variable assignment), in each step the search process moves to a configuration selected from the local neighbourhood (typically based on heuristics function). This process is iterated until a termination criterion is satisfied. To avoid stagnation of the search process, almost all local search algorithms use some form of randomisation, typically in the generation of initial positions and in many cases also in the search steps. Local search algorithms cannot be used to show that a CSP does not have a solution or to find an optimal solution. However, such algorithms are often effective at finding a solution if one exists and can be used to approximate an optimal solution.

The *state of art of the constraint solvers* is mainly presented in the *Constraint Solver Competition* (CSC). At the writing time this event reached its fourth international edi-

tion in 2009.[3] However, in this competition only solvers capable of accepting a CSP with the specific *XML* format, XCSP [35] are admitted. The results are ranked based on the number of solved instances and ties are broken by considering the minimum total solution time. Based on this criteria of ranking, Figure 2.4 shows the results of the 2009 competition. Among the participating solvers *pcs*, *abscon*, *choco*, *sugar* and *mistral* are winners in one or more category. In the specific *mistral* and *sugar* are tied winner with both 8 leading categories over 21, then *choco* and *abscon* won 2 and *pcs* 1.

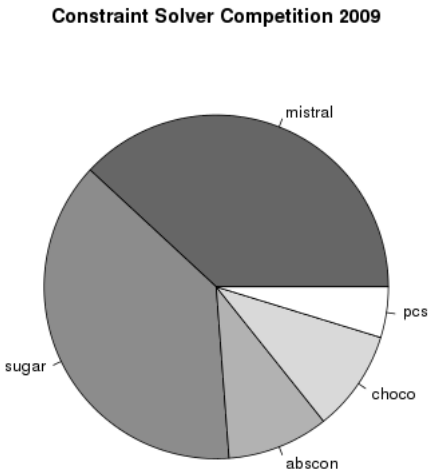


Figure 2.4: The winner of the 2009 constraint solver competition.

Unfortunately no direct ranking is provided on the "speed" of the solvers or at least on the ratio between *CPU time* and *number of solved instances*.

As well as constraint solvers, SAT solvers state of art is presented in different competition like the International SAT Competition [4] and the SAT Race. At the writing time the last competitions held are in 2009 for the SAT competition [5] and 2010 for the SAT Race. [6] The variety of SAT solvers participating at these competitions are high and the categories of competition are split on type of instances, representation of problems or type of execution.

[3]http://www.cril.univ-artois.fr/CPAI09/
[4]http://www.satcompetition.org/
[5]http://www.satcompetition.org/2009
[6]http://baldur.iti.uka.de/sat-race-2010/index.html

From the 2009 SAT competition, 9 are the categories in which the solvers are tested. Among the competing solvers, the portfolio solver SATzilla won 3 out of 9, *clasp* 2 categories and *precosat* , *glucose*, *TNM* and *March hi* one.

From the 2010 SAT race, among the participating solvers, the winners of the three competitions are *CryptoMiniSat*, *plingeling* and *MiniSat++ 1.1*.

It is interesting to see how SATzilla is employing the best performing solvers of the 2006 SAT race and 2005 SAT competition for winning three categories both in the 2007 and 2009 SAT competition. This portfolio of SAT solvers will be described in Section 2.3.1.

2.3 Algorithm portfolio

Over the past decade there has been a significant increase in the number of solvers that have been deployed for solving constraint satisfaction problems. It is recognised within the field of constraint programming that different solvers are better at solving different problem instances, even within the same problem class [12]. It has been shown in other areas, such as satisfiability testing and integer linear programming that the best on average solver can be outperformed by a portfolio of possibly slower on average solvers, examples are respectively SATzilla [43] and this work [23]. The same philosophy is applied also to some latest machine learning algorithms, like the previously mentioned *AdaBoost* (Section 2.1.3). Instead of using a single classifier, a series of "weak" classifiers are combined together in a way to reach a consensus. The portfolio selection process is usually performed using a machine learning technique based on feature data extracted from constraint satisfaction problems.

Portfolio solvers are often distinguished by the following categorization:

- parallel, where all the solvers all run in parallel;
- sequential, where a solver waits for the previous one to finish, before starting its computation;
- partly sequential, an hybrid version of the previous categories.

Two portfolio solvers are involved in the hereby presented work. CPHydra is the protagonist portfolio solver. From which, we extract the features to manipulate. The second one is SATzilla. Even if it is a portfolio of SAT solvers, it is useful to establish the goodness of CPHydra features by means of comparison.

2.3.1 SATzilla

SATzilla [43] is an automated approach for constructing per-instance algorithm sequential portfolio for SAT problems. SATzilla uses machine learning techniques, such as linear regression, to build run time prediction models based on features computed from instances.

In the specific, the original version of SATzilla uses a very specific strategy for building a sequential portfolio. The part of construction is done offline. After selecting a set of instances believed to be representative of some underlying distribution, SATzilla selects a set of candidate solvers that are expected to perform well on the data distribution. The appropriate features are then selected on these problem instances. This operation is usually done with the help of knowledge from a domain expert. The features set are calculated on a training set of instances, determining also the real running time for each algorithm candidate.

In the next phase SATzilla identifies one or more solvers that have to be used as pre-solvers. Such solvers are used to eliminate those problems that are solved in a short amount of time and will be launched before the feature computation. Using a validation dataset, SATzilla then determines the backup solver. This solver is the one which achieves the best performances on those instances that are not solved by the pre-solvers. In absence of instances that go timeout or produce error, the best single solver known so far is selected as backup solver. Then a model containing the runtime predicted for each solver is constructed and SATzilla selects the best subset of solvers to use in the final portfolio.

In the online phase, SATzilla runs the pre-solvers for a limited amount of time; if the pre-solvers succeed the process end up here but if the pre-solvers fail to solve the problem then it computes the features of the instance just run. If, once again, these features are not calculated because of errors or timeout, then the backup solver is used for solving the instance. If instead the features are calculated, then the runtime algorithm is predicted using a simple variant of linear regression and finally, once the best predicted is selected, this is run. In the International SAT Competition 2009, SATzilla won all three major tracks of the competition.[7]

2.3.2 CPHydra

CPHydra [30] is sequential portfolio solver for constraint satisfaction problems developed at 4C (Cork Constraint Computation Centre) that uses a CBR machine learning algorithm to select solution strategies for constraint satisfaction. A CBR methodology

[7] http://www.satcompetition.org/2009/

performs well in decision making as reported in [11]. CPHydra was the overall winner of the 2008 Constraint Solver Competition. Its strength relies on the combination of machine learning methodology, case based reasoning (CBR), with the idea of partitioning CPU-Time between components of the portfolio in order to maximise the expected number of solved problem instances within a fixed time limit. Mainly because it has been built for the 2008 competition, CPHydra uses a portfolio with three solvers (*abscon*, *choco* and *mistral*) and it does not exploit the latest results of the competition, where *sugar* won the same number of categories as *mistral*. However, comparing *mistral* to the other solvers will clearly establish that it is the fastest solver in the portfolio.

In the retrieval phase of the CBR, CPHydra uses a *k-nearest neighbor* algorithm that returns the set of solving times for each of the k most similar problem instances found in the base case along with their similarities to the submitted one. In order to define how to evaluate the similarities between the cases, CPHydra uses a classic Euclidean distance between the features of the problems. In situation of equality, tied cases are returned in the k neighbour. In the reuse phase the runtime of the k most similar cases are used to generate a solver schedule. For each problem submitted, γ are the similar cases returned. For a given solver s and a time point $t \in [0 \dots 1800]$, we define $C(s,t)$ as the subset of γ solved by s given at least time t. The goal of the scheduler is then to maximise $C(s,t)$, for each solver. This goal is then optimised weighting the input case by the Euclidean distance of each similar case.

During the revision phase a solution is evaluated and validated running each solver for the amount of time decided by the scheduler. If only one solver solves the instance within the time slot, then the schedule is considered a success. A revision of the solution is not possible in a competition scenario, where there would not be enough time for running each solver. In the retention phase, as well, there is not enough time for building the complete new case. Such an operation requires to run every solver till a solution is reached and in competition scenario this is not possible.

The experiments and the 2008 competition results showed that CPHydra is implementing a winning strategy. In the 2008 competition, CPHydra was the overall winner.[8]

[8] http://www.cril.univ-artois.fr/CPAI08/

Chapter 3

Learning from problem features

Learning is a generic concept that can be applied to many fields. Hereby, the learning concept will be applied to CSPs for information extraction. This information is going to be useful for solving CSPs in the context of a portfolio solver, as will be explained in Chapter 4.

Next we will first introduce the dataset used. Then we will analyse how the portfolio is behaving on this dataset. We will tackle the solving times of each solver for building a group classifiers. With these classifiers we will perform two kinds of tests: one to test different features set of CPHydra and the other one to compare the CPHydra features to those of SATzilla. The latter test will establish how reliable our classifiers are.

In order to avoid confusion, in the next sections we will talk about instance or problem, indicating a CSP instance.

3.1 The international CSP competition dataset

The comprehensive dataset of CSPs used is based on the various instances from the annual International CSP Solver Competition from 2006-2008. Overall, there are five categories of benchmark problems in the competition:

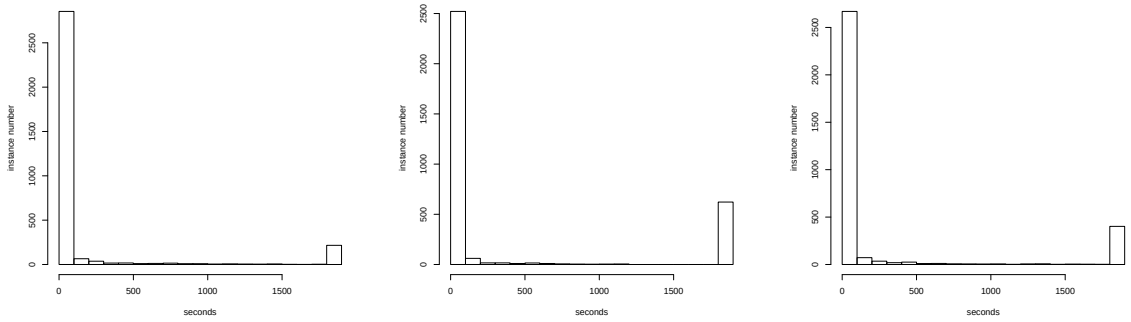
- 2-ARY-EXT instances involving extensionally defined binary (and unary) constraints;
- N-ARY-EXT instances involving extensionally defined constraints, at least one of which is defined over more than two variables;
- 2-ARY-INT instances involving intensionally defined binary (and unary) constraints;

- N-ARY-INT instances involving intensionally defined constraints, at least one of which is defined over more than two variables;
- GLB instances involving any kind of constraints, including global constraints.

The competition stated that a problem would have to be solved with a time out value of 1800 seconds. Once the time frame is passed, the problem is considered not solved. So, in the dataset considered, it is clear to see whether or not a solver finds a solution before the cut off. Another important remark is that, at the competition time, the problems that are not solved by the portfolio within 1800 seconds of cut off, are removed from the dataset. In total, the dataset contains approximately more than 4000 instances across these various categories. Later we will restrict this dataset to a subset of 3293 for reasons that we will clarify.

3.2 Portfolio solving time analysis

Based on the three solvers of the portfolio used in the 2008 CSP Solver Competition variant of CPHydra, it is interesting, as a first step, to present the runtime distributions. Figures 3.1(a), 3.1(b), 3.1(c), respectively *mistral*, *choco* and *abscon* solving time distributions, are introducing their distributions. Each runtime distribution is specified as a histogram of the frequency (y-axis) of a given runtime (x-axis). The purpose of doing this is to show for every single solver in the portfolio, the number of instances of the dataset solved in the given time windows. This can highlight the fact that there are many instances for which one of the solvers finds a solution quickly, while another solver struggles to solve them. However, it is not the case that each solver finds that the same instances are either easy or hard because. For instance, *abscon* can solve some instances in an easier way than *mistral* and the situation is the other way around in some other instances. This is a good hint for developing what will be one of the first classifiers, which however will be more useful in the scheduling part: a classifier which will try to forecast the fastest solver in the portfolio.



(a) Runtime distribution for Mistral.

(b) Runtime distribution for Choco.

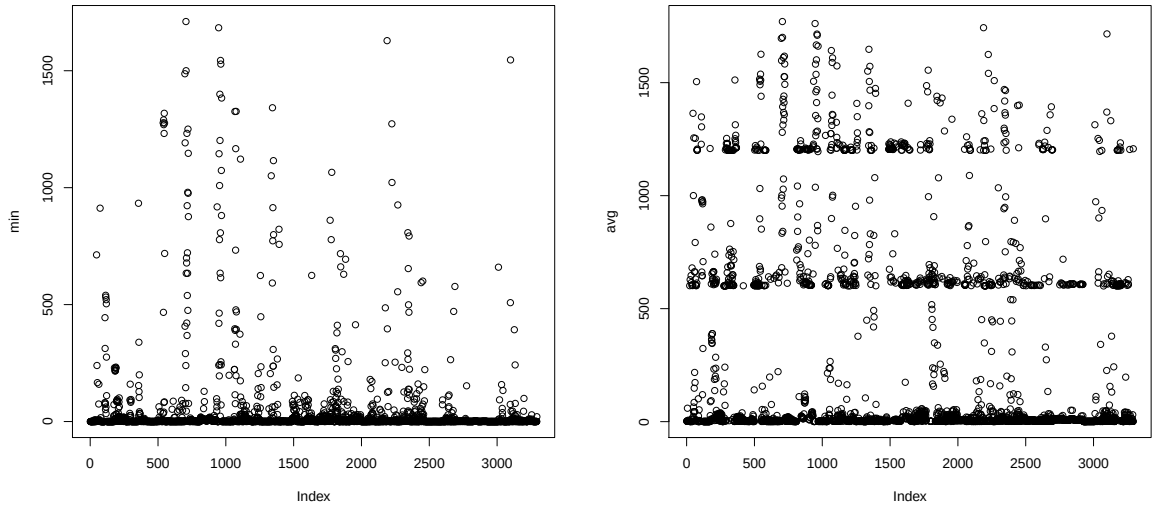
(c) Runtime distribution for Abscon.

Figure 3.1: The performance of each solver in the portfolio on the dataset.

We note in Figure 3.1(a) that *mistral* seems to be the fastest solver only analysing the bars of the plot. In fact it is the one with the higher bar on the left side and lowest bar in correspondence of the timeout value.

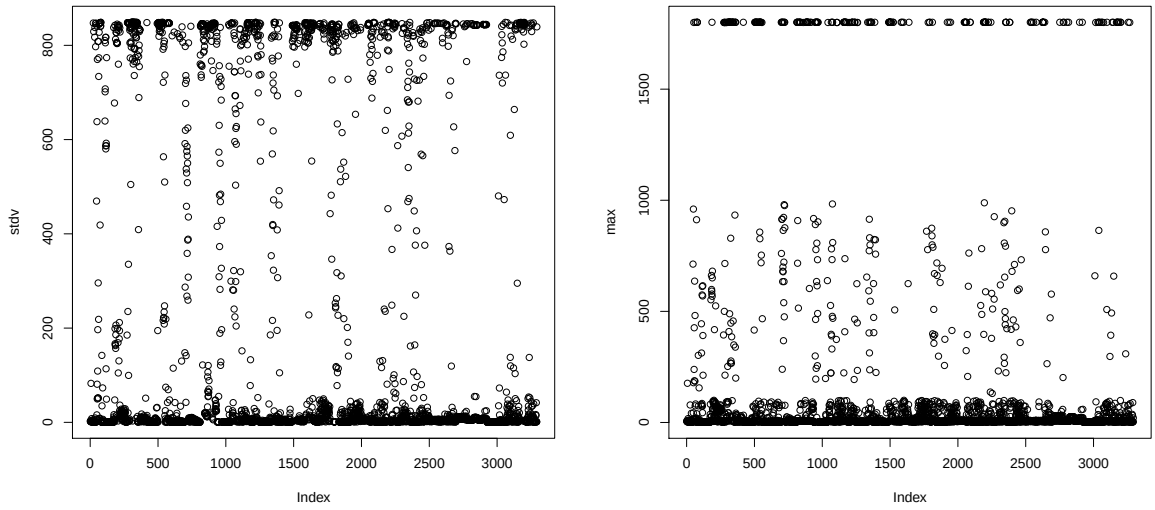
After showing how the single solvers of the portfolio are behaving, it is interesting to address directly the portfolio itself. For each instance, we extract the minimum, maximum and the average runtime of the instances given by the three solvers in the portfolio, as well as the standard deviation. Analysing such data could highlight any kind of clustering or distribution patterns showed by the combination of the solvers. A cluster or a distribution pattern in the plot, would manifest that a good machine learning classification could be created.

In the plots below (Figures in 3.2) we show the output distribution of data analysed. All the functions applied show a natural clustering of the instances, except the minimum. In fact the minimum plot (Figure 3.2(a)) shows a distribution that does not indicate any immediate clustering or possible classification.



(a) The minimum runtime of the portfolio on each instance.

(b) The average runtimes of the portfolio on each instance.



(c) The standard deviation runtimes of the portfolio on each instance.

(d) The maximum runtimes of the portfolio on each instance.

Figure 3.2: Average, standard deviation and maximum runtimes of the portfolio.

While considering average, standard deviation and maximum, it is very clear that there are clustering on instances and very precise lines. In Figure 3.2(b) there are three evident

classes, one starting from 0 seconds to around 600 seconds, the second one starting from 600 to 1200 seconds and the third one from 1200 to 1800 (timeout). These specific values are not randomly appearing, but they are result of the average operator. Moreover the plot shows that the values are concentrated on the boundaries. The first class is close to zero, meaning that all the solver are fast on average. The second class is close to 600, meaning that there might be one solver of the portfolio going timeout while the rest are fast. The third class of instances are grouped close to 1200, meaning that there might be two solvers going timeout and one solving the problem.

In Figure 3.2(c) it is clear that the majority of the values are either concentrated to the top or to the bottom, with few middle values. This suggests two clusters. Therefore we understand that either a problem is close to its average solution time or the portfolio has very fast and very slow solvers. In Figure 3.2(d) it is even clearer than the previous plot that there are two natural clusters. One below the 1500 seconds and the other one on the top at the timeout value. This implies that in the worst either a problem is solvable within 1500 seconds or it is likely to time out.

3.3 Classifiers

Upon the discovery of distribution patterns, the next step is to exploit those properties to create classifiers that are able to convey information about the satisfiability of the problems. In total, six classifiers are developed:

- 3 classes (3C);
- average (AvgC);
- standard deviation (StdvC);
- maximum (MaxC);
- fastest solver (Fs);
- mistral classifier (MistralC).

While the first four are used both for feature comparison and scheduling simulation, Fs and MistralC classifiers are only used in the scheduling.

Table 3.1 summarizes the classifiers and the way they select classes.

Classifier	Class	Meaning
3C	easy for all	if all the solvers in the portfolio solves the problem within $10s$
	at least one easy	if at least one solver solves the problem within $10s$
	never easy	if none of the solvers in the portfolio solves the problem within $10s$
AvgC	easy	portfolio average solving time (p_{avg}), $p_{avg} \leq 600s$
	medium	$600s < p_{avg} \leq 1200s$
	hard	$p_{avg} > 1200s$
StdvC	low	portfolio standard deviation solving time (p_{stdv}), $p_{stdv} \leq 100s$
	high	$p_{stdv} > 100s$
MaxC	easy	portfolio maximum solving time (p_{max}), $p_{max} \leq 1500s$
	hard	$p_{max} > 1500s$
Fs	abscon	$a_t < c_t \wedge a_t < m_t$
	choco	$c_t < a_t \wedge c_t < m_t$
	mistral	$m_t < c_t \wedge m_t < a_t$
MistralC	easy	mistral solving time (m_t), $m_t \leq 10s$
	medium	$10s < m_t \leq 1799s$
	hard	$m_t > 1799s$

Table 3.1: Classifiers and their classes.

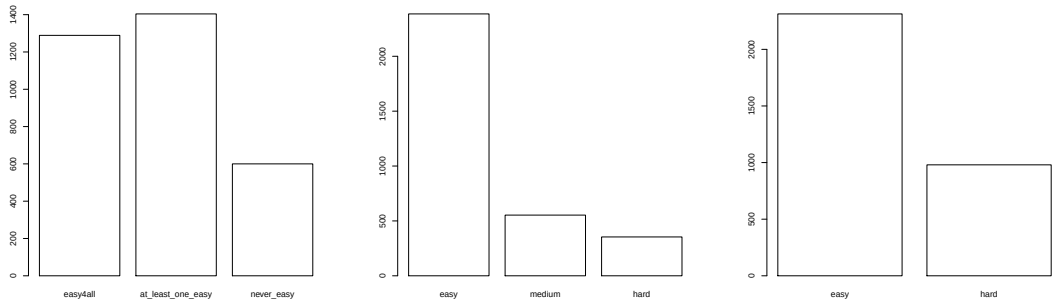
The 3C classifier is the most important one among the group since it is the one that better separates the problems with respect to the portfolio. 3C aims to predict where the portfolio is able to solve an instance easily with all the solvers, with at least one or never easily. The AvgC classifier aims to give an indication on the average solving time necessary in the portfolio to solve the examined instance. Along with the average there is often the standard deviation, a measure of spread of the portfolio around the average value. A high value of standard deviation could mean that in the portfolio there are fast or slow solvers, while a low value could mean that the portfolio has a solving time close to the average. The StdvC classifier measures the standard deviation/spread using two classes: low and high. The MaxC classifier employs two classes with the goal to distinguish between those portfolios that have at least one solver which is likely to go time out and those portfolios which does not have it at all. The Fs classifier simply aims to predict which solver is the fastest in the portfolio. The MistralC classifier is built for test purposes and works in this way: if an instance is solved within 10 seconds, it is classified easy. If it times-out, it is classified hard. If it is in the middle of the previous cases, it is classified medium. The classifier MistralC is developed because *mistral* is the best solver of the portfolio as well as one of the top solvers in the 2009 International CSP competition, like mentioned in Section 2.2.4. Later we will also present how the *mistral* solver is the fastest solver for 77,89% of the dataset considered. MistralC will be useful once we will compare the performance of solving the CSPs with a classification based portfolio or using only the *mistral*, the best solver of the portfolio.

3.4 Classifiers output distribution

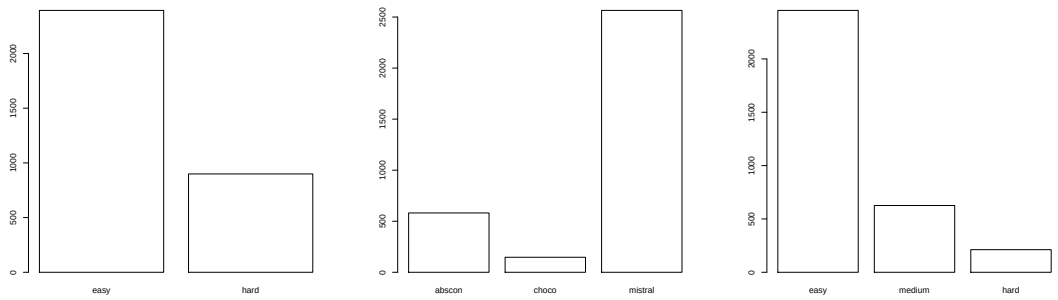
Having to deal with unbalanced data set, is not a good feature for machine learning algorithms [31]. Simply because if, on a two classes problem, one of the classes is chosen by the 98% of the instances, then the optimal strategy will classify all the new coming instances on the most selected class. This is a generalizing behaviour of the algorithms that are trying to reach the best accuracy possible and is not an advantage in case we are interested also in the other 2% of the instances.

Usually when those problems are present upsampling and downsampling are the first techniques suggested. The former adds new instances replicating from the minority set, while the latter is reducing the instances set, eliminating instances from majority set.

In the following series of Figures (3.3) we will analyse how the instances are spreading over the different classes.



(a) The 3C classifier classes distribution. (b) The AvgC classifier classes distribution. (c) The StdvC classifier classes distribution.



(d) The MaxC classifier classes distribution. (e) The Fs classifier classes distribution. (f) The MistralC classifier classes distribution.

Figure 3.3: Classifiers classes distribution.

Three classifiers seem to show high level of imbalance, namely AvgC Fs and the testing classifier MistralC. However, only Fs (Figure 3.3(e)) and MistralC (Figure 3.3(f)) are in unbalanced condition where the classes 'choco' and 'hard' represent the 4.46% and 6.43% of the set. This issue is due to the nature of the problems we are treating. In fact, in Figure 3.1(a) we showed the solving time distribution of *mistral*. The plot represent perfectly the classifier classes distribution of MistralC. The majority of the instances are solved easily while a small amount is slower or unsolvable.

Techniques such as upsampling and downsampling are not contemplated in this context. The first one would influence the MLA to consider more important those problems features of CSPs that are replicated. The second one would reduce the instances set to such a small number of problems that would make training useless. This problem rests unsolved, nevertheless the experimental results showed in Section 3.6.3 give us a good reason to think that, despite that the unbalance issue our approach is still reliable.

The Fs classifier has an interesting result within. It allows us to prove, as previously stated, the dominance of *mistral* as the fastest solver of the portfolio. In fact 77.89% of the instances are solved by *mistral* with the best solving time of the portfolio. Classifiers like StdvC (Figure 3.3(c)) and MaxC (Figure 3.3(d)) are following the trend of the time distribution of the solvers without showing an impressive imbalance. Last but not least, the 3C (Figure 3.3(a)) classifier shows a good distribution among the classes. The first two classes are gathering the majority of the instances. This gives the hint that 3C could be good to distinguish first the very easy instances for the whole portfolio, then those that are fast for some solvers and then taking care of the hard ones with further methods.

3.5 Features set

The choice of the instance features has a significant impact on the performance of the classifiers. If a feature is expressive of the problem hardness, like the arity of a constraint, this particular characteristic can influence the MLA to perform a better classification. The list of features used by the portfolio solver CPHydra are analysed briefly in Section 3.5.1. Furthermore, in Section 3.5.3, a new group of features is added to the existing CPHydra features. For establishing the goodness of them, it is necessary to compare them with another good portfolio algorithm solver. For this comparison test we use SATzilla features, described in Section 3.5.2 .

3.5.1 CPHydra features

An aspect of CPHydra not covered in the introduction of the portfolio solver is its features set. CPHydra portfolio solver employs 36 features. A part of the features are calculated by running the solver *mistral* and using the same solver for extracting information statically. The rest are syntactic features. Here we list all the features, numbered 1-36 and grouped by how they are represented.

Dynamic features:
1-4: log average variables weight, log nodes, log propagation, log standard deviation variables weight.

Logarithmic features:
5-16: bits, boolean, constants, constraints, extra bits, extra boolean, extra ranges, extra values, lists, ranges, search variables, values

General purpose features:
17-18: max arity, all-different constraints

Percent features (%):
19-28: all-different, avg continuity, cumulative, decompose predicate, element, extensional, gac predicate, global, min continuity, weighted sum

Perten features ($\times 10$):
29-34: average predicate arity, avg predicate shape, average predicate size, binary ext, large ext, naryext

Square root features:
35-36: average domain size, max domain size

The dynamic features are obtained by launching *mistral* for each instance, with a time limit of 2 seconds. The attributes that are saved are the average and standard deviation of the variables weight. Such a measure is modified by *mistral* every time a failure in the searching phase of the solution occurs. Along with those two values, there are the number of nodes created and the number of propagations done.

The logarithmic category groups all those kind of features in which the function log_2 is applied and mostly these are the one calculated using *mistral* statically. Some of the

features are related to how *mistral* represents the problem for solving it. In this subset it is possible to find the number of bits, booleans, constants, extra bits, extra boolean, extra ranges, extra values, lists, ranges and values. Feature number 8 measures the number of constraints, while feature number 15 counts the number of variables that *mistral* will have to search on. In the general purpose group there are two features which are not mathematically transformed, the max arity of the CSP and number of *all-different* constraints present.

There is then a substantial group of features expressed in percentage. First of all there are the percentage of extensional and global constraints respect to the total number of constraints. Then we also count the percentage of decomposed predicates and the non-decomposed (gac predicates) on the number of constraints. For predicate we mean mathematical, logic formulas that can involve inequalities, equalities or simply logic operations. There is the percentage of all-different, element, cumulative and weighted sum constraints with respect to the global constraints. These are all different types of global constraints. The percentage of domain continuity measure how the domain interval is covered by the domain values assumed. If the percentage is high it means that the continuity of values is good; if the percentage is low it means that the values are spread in the domain with large gaps between one value and another one. We measure the percentage average and minimum of the domain continuity.

In the per ten features group we store information like the average predicate arity, size and shape, expressed as scaled from 0 to 10. The shape is defined to be the average of the score of the predicates, knowing that each operation within a predicate has a different score (i.e. one default, 5 for a mul, 100 for a pow). Moreover there are the features that capture the per ten value of the binary, n-ary and large constraints (with arity ≥ 5) with respect of the extensional constraints. The squared root features group contains information about the maximum and average domain size.

3.5.2 SATzilla features

SATzilla is using a restrained set of features introduced by Nudelman [29]. From the 84 mentioned in the paper, SATzilla cut out a number of computationally expensive features, limiting the computation time for each of the local search and other features extractions. The total feature computation time per instance was limited to 60 CPU seconds. After eliminating some features that had the same value across all instances and some that were too unstable given only 1 CPU second of local search probing, SATzilla ended up using the 48 features summarized in the following list [43].

Problem Size Features:
1. **Number of clauses:** denoted c.

2. **Number of variables:** denoted v.
3. **Ratio:** c/v .

Variable-Clause Graph Features:
4-8. **Variable nodes degree statistics:** mean, variation coefficient, min, max and entropy.
9-13. **Clause nodes degree statistics:** mean, variation coefficient, min, max and entropy.

Variable Graph Features: 14-17. **Nodes degree statistics:** mean, variation coefficient, min and max.

Balance Features:
18-20. **Ratio of positive and negative literals in each clause:** mean, variation coefficient and entropy.
21-25. **Ratio of positive and negative occurrences of each variable:** mean, variation coefficient, min, max and entropy.
26-27. **Fraction of binary and ternary clauses**

Proximity to Horn Formula:
28. **Fraction of Horn clauses**
29-33. **Number of occurrences in a Horn clause for each variable:** mean, variation coefficient, min, max and entropy.

DPLL Probing Features:
34-38. **Number of unit propagations:** computed at depths 1, 4, 16, 64 and 256.
39-40. **Search space size estimate:** mean depth to contradiction, estimate of the log of number of nodes.

Local Search Probing Features:
41-44. **Number of steps to the best local minimum in a run:** mean, median, 10th and 90th percentiles for SAPS.
45. **Average improvement to best in a run:** mean improvement per step to best solution for SAPS.
46-47. **Fraction of improvement due to first local minimum:** mean for SAPS and GSAT.
48. **Coefficient of variation of the number of unsatisfied clauses in each local minimum:** mean over all runs for SAPS

3.5.3 CPHydra added features

It was immediately clear that two CPHydra features (7: `log_costants` and 12: `log_extra_values`) were useless, since on each instance, their value was always '-1', which means not calculated. They were consequentially removed from the feature set.

Inspired by the work presented by Nudelman [29], which is used also in SATzilla, we extracted some new features from the problems, modifying the source code CPHydra in order to calculate them. The first simple feature that is added is the ratio of number constraints over number variables and the reciprocal ratio, number of variables over number of constraint. Then more elaborate features were considered. For every instance we considered the variable graph (vg), where every variable is represented as a node and an edge connects two nodes if only if they occur together in at least one constraint.

Finally we considered the variable-constraints graph (vcg) which has a node for every variable or constraint and an edge between a variable and a constraint if only if the variable is involved in that constraint. From those features over the graphs we extracted the average and standard deviation of the nodes degree and then applied the logarithm. All of these new feature were expressed by means of logarithm in base 2.

> **extended features set:**
> 37-44: ratio, reciprocal ratio, log vg average, log vg standard deviation, log vcg average constraint, log vcg average variable, log vcg standard deviation constraint, log vcg standard deviation variable.

3.6 Experimental results

Two types of experiments using our classifiers are conducted aiming to three different purposes. The first one is focused on different features sets of CPHydra. In Section 3.6.1, different combinations of features set will be tested to establish if some configuration is better than another one. The second type of experiments aims to compare CPHydra features to SATzilla ones and a mixture of the two. These experiments satisfies two goals of ours: decreeing the supremacy of CPHydra features over SATzilla one; and establishing that our classifiers are reliable, reaching satisfying values of accuracy and κ statistic.

In each experiment, the Weka Experimenter framework [17] is used using a variety of MLA by classification purposes. A 10-fold cross validation is performed and the performance of each classifier, on each representation and on each classification task is measured in terms of the classification accuracy and the κ statistic.

Differences in performance are tested for statistical significance using a paired t-test at a 95% confidence level. Usually the results are reported in tables and the feature set expressed in the first column is used as a baseline. In each table, values that are marked with a $\circ$ represent performances that are statistically significantly better than the baseline, while those marked with a $\bullet$ represent performances that are statistically significantly worst.

3.6.1 Comparison of different CPHydra feature sets

The purpose of this section is to show the results of a series of classification run on different feature sets. So we will first introduce how the different feature sets are formed. Then we will show how long does the features calculation take and finally we presents the tables highlighting the comparison from the accuracy and κ statistic point of view.

The feature sets to test involve the following list of attributes variation. Each feature is expresses as $id : name$.

> **features involved in the different features sets:**
> 7:constants, 12:extra values, 37:ratio, 38:reciprocal ratio, 39:log vg average, 40:log vg standard deviation, 41:log vcg average constraint, 42:log vcg average variable, 43:log vcg standard deviation constraint, 44:log vcg standard deviation variable

Table 3.2 explains the feature sets composition, considering that the features not specified in the column are kept fixed in each set. The *norm* column shows whether or not all the features are normalized by the variable size and added to the features set. The feature sets names in the first column correspond to the amount of features used.

Feature set	features IDs										
	7	12	36	37	38	39	40	41	42	43	norm
42	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗
34	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
36	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
38	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗
40	✗	✗	✓	✓	✗	✗	✓	✓	✓	✓	✗
84	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
original	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗

Table 3.2: The classifiers used for comparing the different feature set.

While in the main feature set comparison all the previous feature sets are used, for showing the time elapsed in the features calculation we will not. It is necessary to show the bigger, the smaller and the baseline feature set. The time elapsed varies from machine to machine, so what will be important are the differences between the times registered.

Feature set	Elapsed time (sec)	
	no dynamic	with dynamic
34	1019	36549
original	1022	36650
42	1080	36729
84	1087	36768

Table 3.3: Time elapsed for the CPHydra features calculation.

We notice in Table 3.3 that the time elapsed for calculating the features does not increase too much as the number of features increase. The dynamic features are the real reason why the calculation takes so long, in fact without them the calculation would be almost 34 times faster. The explanation relies in the fact that for every problem, *mistral* is executed for a maximum time of 2 seconds in order to be able to extract the dynamic features.

In Table 3.4 we enlist the MLA that are used for the upcoming test. The choice is limited to three because this test does not need any wide range experiment on different type of algorithms. *J48* is employed as baseline *RandomCommittee* and *RandomForest* will demonstrate that they are the reaching the highest values of accuracy and κ statistic.

MLA	Toolkit	Implementation
Decision Tree	WEKA	J48
Meta Classifier	WEKA	Random Committee + Random Tree
Decision Tree	WEKA	Random Forest

Table 3.4: The classifiers used for comparing the different feature sets.

From Table 3.5 to 3.12, a series of tests on classifiers 3C, AvgC, StdvC, MaxC are reported, analysing first accuracy and then κ statistic for each feature set. The features set 42 is used as baseline. In each table, values that are marked with a $\circ$ represent performances that are statistically significantly better than the 42 features set, while those marked with a $\bullet$ represent performances that are statistically significantly worse.

Table 3.5: Feature sets comparison based on the 3C classifier: accuracy.

MLA on 3C	42	34	36	38	40	84	original
trees.J48	83.83	83.04	83.15	83.63	83.64	82.49 •	83.04
trees.RandomForest	85.34	84.86	84.86	85.12	85.27	84.28	84.72
meta.RandomCommittee	84.99	84.80	84.67	85.20	85.37	84.59	84.85

◦, • statistically significant improvement or degradation

Table 3.6: Feature sets comparison based on the AvgC classifier: accuracy.

MLA on AvgC	42	34	36	38	40	84	original
trees.J48	84.18	84.12	84.21	84.14	84.18	83.96	84.12
trees.RandomForest	85.37	85.53	85.28	85.30	85.08	84.95	85.45
meta.RandomCommittee	85.03	85.45	85.12	85.27	85.17	84.60	85.29

Table 3.7: Feature sets comparison based on the StdvC classifier: accuracy.

MLA on StdvC	42	34	36	38	40	84	original
trees.J48	90.10	90.07	90.02	90.16	89.88	89.89	90.07
trees.RandomForest	90.88	90.83	90.70	90.75	90.94	90.80	90.68
meta.RandomCommittee	91.00	90.75	90.63	90.79	90.90	90.78	90.72

Table 3.8: Feature sets comparison based on the MaxC classifier: accuracy.

MLA on MaxC	42	34	36	38	40	84	original
trees.J48	89.61	89.85	89.60	89.75	89.68	89.69	89.85
trees.RandomForest	90.35	90.56	90.42	90.24	90.35	90.19	90.40
meta.RandomCommittee	90.28	90.39	90.30	90.20	90.22	90.17	90.42

Table 3.9: Features set comparison based on the 3C classifier: κ.

MLA on 3C	42	34	36	38	40	84	original
trees.J48	0.75	0.73	0.74	0.74	0.74	0.72 •	0.73
trees.RandomForest	0.77	0.76	0.76	0.77	0.77	0.75	0.76
meta.RandomCommittee	0.76	0.76	0.76	0.77	0.77	0.76	0.76

◦, • statistically significant improvement or degradation

Table 3.10: Features set comparison based on the AvgC classifier: κ.

MLA on AvgC	42	34	36	38	40	84	original
trees.J48	0.63	0.63	0.63	0.63	0.63	0.62	0.63
trees.RandomForest	0.65	0.65	0.65	0.65	0.64	0.64	0.65
meta.RandomCommittee	0.65	0.66	0.65	0.65	0.65	0.64	0.65

Table 3.11: Features set comparison based on the StdvC classifier: κ.

MLA on StdvC	42	34	36	38	40	84	original
trees.J48	0.76	0.76	0.76	0.76	0.76	0.76	0.76
trees.RandomForest	0.78	0.78	0.77	0.77	0.78	0.78	0.77
meta.RandomCommittee	0.78	0.78	0.77	0.78	0.78	0.78	0.77

Table 3.12: Features set comparison based on the MaxC classifier: κ.

MLA on MaxC	42	34	36	38	40	84	original
trees.J48	0.73	0.74	0.73	0.74	0.74	0.74	0.74
trees.RandomForest	0.75	0.76	0.75	0.75	0.75	0.75	0.75
meta.RandomCommittee	0.75	0.75	0.75	0.75	0.75	0.75	0.75

In Table 3.13 we summarise the results given previously from Table 3.5 to Table 3.12. Table 3.13 should be read as follows: the baseline set of 42 features is major, equal or minor $(>, =, <)$, in a number of characteristics indicated in each row, compared to the feature set in the first column across twelve tables. Each comparison considers both accuracy and κ statistic. As Table 3.13 suggests, none of the feature sets was able to perform clearly better than the others. It is also clear from the tables that, although there is no feature set that is statistically more significant than the others, the set with 42 features is performing slightly better. Like summarised in Table 3.13, this set is performing slightly better not only compared to the original features set but also to the other ones. A further run of popular techniques like backward elimination or forward selection does not help refining the features set.

Keeping the baseline set composed by 42 features as the official features set for the following experiments has its strength points. First, the slightly improved accuracy even though we did not find any statistical significance. Secondly the idea of introducing

Feature set compared	baseline:42		
	$>$	$=$	$<$
original-accuracy	7	0	5
original-κ	4	7	1
34-accuracy	7	0	5
34-κ	2	7	3
36-accuracy	8	0	4
36-κ	4	8	0
38-accuracy	8	0	4
38-κ	2	8	2
40-accuracy	6	2	4
40-κ	2	8	2
84-accuracy	11	0	1
84-κ	5	6	1

Table 3.13: Summary of percentage major, equal or minor $(>, =, <)$ between the baseline and the other datasets.

more complete features like those based on graphs and, at the same time, removing two of them useless.

3.6.2 Comparison between CPHydra and SATzilla features sets

Three alternative feature descriptions are compared in this second experiment. These are denoted as CPHydra, Hylla, and SATzilla, in the tables. CPHydra has the new expanded set composed of 42 features. For being able to calculate the 48 features of SATzilla a conversion of CSPs to SAT was required. Unfortunately, as mentioned in [34], this is a process that can take a long time. Indeed it took a long time, used a big amount of memory space and eventually it was not possible to calculate the features of some instances. Because of these calculation issues, after the CPHydra and SATzilla features calculation the dataset shrank down to 3293 instances. These instances are then used for all the three features set.

The solver *sugar* provides the utility of encoding a CSP into a SAT problem. This utility was exploited for having our instances encoded into SAT one and being able to run an executable program, provided by the creator of SATzilla, which extracts the SATzilla features from the SAT problems. A third feature set is formed after the previous one by simply merging the two features set CPHydra and SATzilla. This is called Hylla.

The performances are compared using various MLA presented in Table 3.14, on each of the four classifier (3C, AvgC, StdvC and MaxC). In this case the choice of MLA presented is bigger because there is the will of trying to see how the different feature set are behaving with the respect to different learning categories previously not used, like *NaiveBayes* or *One-rule*. *ANN* is not included because of slowness and difficulties in tuning. Originally *MultiBoostAB* was tested in combination with *J48* but the computation time was unacceptably high that it would make impossible to test such configuration for scheduling purposes. So, like for *RandomCommitte*, *RandomTree* is used as weak learner for *MultiBoost*.

MLA category	Toolkit	Implementation
Decision Tree	WEKA	J48
k-NN	WEKA	IBk
Meta Classifier	WEKA	MultiBoostAB + RandomTree
Meta Classifier	WEKA	Random Committee + RandomTree
Naive Bayes	WEKA	Default
Decision Tree	WEKA	Random Forest
Rules	WEKA	One-R

Table 3.14: The classifiers used for evaluation.

In the following tables CPHydra feature set is used as a baseline. The symbol $\circ$ then, represents performances that are statistically significantly better than CPHydra, while those with a $\bullet$ represent performances that are statistically significantly worst. Each table is ordered according to the accuracy on CPHydra.

Table 3.15: Classification accuracy and κ-statistics for the 3C classifier.

MLA	CPHydra	Hylla	SATzilla	CPHydra	Hylla	SATzilla
trees.RandomForest	85.34	85.14	82.39 •	0.77	0.77	0.72 •
meta.MultiBoostAB	85.04	81.58 •	81.66 •	0.76	0.71 •	0.71 •
meta.RandomCommittee	84.99	85.44	82.57 •	0.76	0.77	0.73 •
trees.J48	83.83	82.52 •	79.70 •	0.75	0.73	0.68 •
lazy.IBk	83.69	83.53	78.71 •	0.74	0.74	0.67 •
rules.OneR	72.89	72.89	69.16 •	0.57	0.57	0.51 •
bayes.NaiveBayes	61.70	62.74	52.49 •	0.37	0.44 ∘	0.31 •

∘, • statistically significant improvement or degradation over CPHydra .

Table 3.16: Classification accuracy and κ-statistics for the AvgC classifier.

MLA	CPHydra	Hylla	SATzilla	CPHydra	Hylla	SATzilla
trees.RandomForest	85.37	85.53	84.42	0.65	0.65	0.62 •
meta.RandomCommittee	85.03	85.32	84.25	0.65	0.65	0.62
trees.J48	84.18	83.35	82.42 •	0.63	0.61	0.58 •
lazy.IBk	83.45	83.13	81.39 •	0.62	0.61	0.57 •
meta.MultiBoostAB	83.06	82.50	80.74 •	0.61	0.60	0.56 •
rules.OneR	78.97	78.97	75.75 •	0.45	0.45	0.34 •
bayes.NaiveBayes	64.96	53.81 •	41.69 •	0.25	0.22	0.12 •

∘, • statistically significant improvement or degradation CPHydra:

Table 3.17: Classification accuracy and κ-statistics for the StdvC classifier.

MLA	CPHydra	Hylla	SATzilla	CPHydra	Hylla	SATzilla
trees.RandomForest	91.03	91.14	90.15	0.78	0.78	0.76
meta.RandomCommittee	91.02	91.08	90.17	0.78	0.78	0.76 •
meta.MultiBoostAB	90.70	88.28 •	87.31 •	0.78	0.72 •	0.70 •
trees.J48	90.28	89.46	88.22 •	0.77	0.75	0.72 •
lazy.IBk	89.88	89.67	87.90 •	0.76	0.75	0.71 •
rules.OneR	81.43	81.41	77.70 •	0.54	0.54	0.42 •
bayes.NaiveBayes	71.59	65.73 •	55.39 •	0.32	0.33	0.19 •

∘, • statistically significant improvement or degradation over CPHydra .

Table 3.18: Classification accuracy and κ-statistics for the MaxC classifier.

MLA	CPHydra	Hylla	SATzilla	CPHydra	Hylla	SATzilla
trees.RandomForest	90.35	90.70	89.90	0.75	0.76	0.74
meta.RandomCommittee	90.28	90.64	90.10	0.75	0.76	0.74
trees.J48	89.61	89.08	87.99 •	0.73	0.72	0.69 •
lazy.IBk	89.18	89.00	87.87 •	0.73	0.72	0.70 •
meta.MultiBoostAB	88.42	88.32	86.95 •	0.71	0.71	0.67 •
rules.OneR	82.98	82.98	77.87 •	0.54	0.54	0.38 •
bayes.NaiveBayes	71.37	67.30 •	54.26 •	0.31	0.34	0.18 •

∘, • statistically significant improvement or degradation over CPHydra .

In summary, classification accuracies of CPHydra are in majority all higher across all the results, except some cases like some of the *Random Forest* or *Random Committee* results in which no statistically significance is reached. In AvgC, StdvC and MaxC classifiers, all the algorithms except the aforementioned, are showing statistical significance of CPHydra on SATzilla, both on accuracy and κ statistic. In two specific cases this pattern is broken, in AvgC *Random Forest* κ and in StdvC *Random Committee* κ are statistically significantly better than SATzilla. The 3C classifier shows the best results in terms of both accuracy and κ, where all the algorithms used reach the statistical significance of CPHydra on SATzilla. *Multi Boost* and *J48* are reaching

higher value on CPHydra respect to Hylla.

3.6.3 Reliability of classifiers

We showed already in Tables 3.15, 3.16, 3.17 and 3.18 that the classifiers presented are reliable. Not considering the low performing MLA, like *One-R* and *NaiveBayes*, 3C and AvgC reach a really good accuracy value around the 85%. While StdvC and MaxC are reaching accuracy of even 90%. All the κ statistics value are significantly high. AvgC scores a 0.65 level of agreement. While the other classifiers are scoring a really good 0.75/0.78 of κ statistic.

In Table 3.19 we introduce the experimental results of the Fs classifier. This classifier has been excluded from the previous tests because of its utilization is to the next phase described in Chapter 4.

Table 3.19: Classification accuracy and κ-statistics for the Fs classifier.

MLA	CPHydra	Hylla	SATzilla	CPHydra	Hylla	SATzilla
trees.RandomForest	89.28	89.35	88.47	0.69	0.69	0.66
meta.RandomCommittee	89.11	89.44	88.11	0.69	0.70	0.65 •
lazy.IBk	88.60	87.60 •	86.10 •	0.69	0.66 •	0.62 •
trees.J48	88.33	88.09	87.45	0.67	0.66	0.64
meta.MultiBoostAB	87.04	87.02	87.99	0.64	0.64	0.66
rules.OneR	82.85	82.66	82.33	0.44	0.42	0.40
bayes.NaiveBayes	52.96	57.52 ∘	37.56 •	0.25	0.29 ∘	0.15 •

∘, • statistically significant improvement or degradation over CPHydra .

The Fs classifiers reaches an high accuracy value around the 89%, with κ statistic around 0.69.

The CPHydra features set thus gives rise to the best overall performance. Moreover the classifiers built on these features are reliable. Based on these promising results, we consider in the next chapter the utility of using these classifiers as a basis for managing how a solver portfolio can be used to solve a collection of CSP instances.

Chapter 4

Scheduling problems based on learning

The usage of a portfolio of constraint solvers might have different challenges, like minimising the CPU time used for solving the instances or maximising the number of instances solved in a specific time frame. In this chapter we consider the challenge of minimising the average finishing time of an instance set. The finishing time of an instance is defined as the time by which it is solved.

The purpose of the following sections is to show that classifiers can help devising simple scheduling rules that will optimise the average finishing time. For this purpose, a couple of simple *Java* simulators, described in Section 4.3, have been built. One simulates the execution of the portfolio on a single processor case and the other one on a multi processor case. We assume that CSP instances to be solver are all ready at our disposal. We then focus on the 3293 instances considered in the previous chapter. This methodology is similar to the one used in the international SAT competitions.

In both simulators a 5-fold cross validation on a randomised dataset is run. The reason of randomising is due to the fact that similar instances are next to each other and by randomising, the range of problems varies giving the possibility of training on heterogeneous instances. The reason of applying a 5-fold cross validation and not a popular 10-fold like used previously, relies on the dimension of the dataset. A 5-fold split allows to simulate the scheduling with 658 or 659 instances while a 10-fold would halve the dataset. A so small number of instances would not be desirable especially for the multi processors experiments. At the increase of the number of processors, the results returned would not be interesting any more.

In the following section we will explain the simulation and the results on the single processor. A section on the simulation on the multiple processor case will then follow

and finally a section about the simulator implemented.

4.1 Single processor case

A single processor scenario is the elementary way to show a simulation of scheduling. However if the classifiers previously introduced would not be perform well, it would be clear from the results of a simple scheduling simulation. The single processor case is a first step that must be solver before advancing in multiple processors scenario.

For describing the modus operandi of this simulation, we will follow the same execution flow of the simulator developed. In a simple simulation run these operation are called:

1. initialization;
2. preparation of data structures;
3. classification;
4. scheduling;
5. statistics record;
6. structures cleaning.

After a generic initialization in the first phase, the 5-fold cross validation is set up in the second phase of preparation of the data structures. A train and a test set are composed for each classifier employed. The purpose is then training each train set and, on the classifier built, classify the instances belonging to the test. The next step is to apply different scheduling rules to evaluate the classifiers performances. The scheduling rules are deepened in the following Section 4.1.1.

The statistical data is then stored for experimental tests treated in Section 4.1.2. In this single CPU case, two statistical measures are considered. Apart from the already known average finishing times, the median finishing time is recorded. The median gives an idea about which value divides in two the finishing time distribution. It can be a good indicator of how much the data is skewed compared to the average.

A final cleaning function is called to erase all the sets and statistics achieved till this point. This operation is crucial if the 5-fold cross validation is run different times. In the specific, in order to stabilize the results, this whole process is repeated for 250 times.

The average and median finishing time is calculated on each result from each fold of cross validation in every executed run.

In all the tests, we employed *Random Committee* as MLA. This choice is justified in Figure 4.10.

4.1.1 Scheduling rules

Minimising the average finishing time can be achieved by solving each instance with an increasing order of difficulty, more precisely, using the known heuristic shortest processing time. In the portfolio context doing this corresponds to ordering the instances accordingly to their degree of hardness and then employing the fastest solver to solve the instance. This section will be devoted to developing heuristic for this purpose which will be refer to as scheduling rules.

Our scheduling rules are divided in three categories:

- oracle based;
- classifier based;
- baseline.

In the first category there are all those rules that are exploiting directly the real run time. They are using an "oracle" to selects the fastest solver. $\texttt{Oracle}_1$ is what can also be defined as the optimal solution. In fact the instances are ordered based on the fastest solving time and run with that. $\texttt{Oracle}_2$ rule orders the instances according to a combination of the whole 3C, AvgC, StdvC, MaxC classifiers. The specific order will be described later in the rules based on classifiers. $\texttt{Oracle}_3$ orders the instances by the 3C classifier. Such order consists in first execute those instances belonging to the first class (*easy for all*), then instances from the second class (at least one easy) and finally the hard instances (*never easy*).

The rules based on classifiers exploits all the five classifiers developed in chapter 3 to order the instances in different way. All the rules, however, use the Fs classifiers to selected the predicted fastest solver. The rules presented in this category are six. The simplest one is $\texttt{Scheduler}_6$, which orders the instances by the 3C classifier (like $\texttt{Oracle}_3$). The $\texttt{Scheduler}_5$ and $\texttt{Scheduler}_4$ order the instances based on two classifiers: 3C and then respectively by MaxC and AvgC to break ties. The scheduling order is showed in Figures 4.1 and 4.2 with the help of a structural figure as a tree. The root node of the tree indicates the first classifier applied. Each branch is result of ordering on the parent classifier and the results are subsets of the original datasets. A node with a single child or

a leaf is a class. While a node with more than one child corresponds to another ordered classifier. If not specified, the scheduling order has to be read from the most left leaf to the most right leaf.

In all the presented scheduling rules, the labels $c1, c2, c3$ correspond to the 3C classes *easy for all, at least one easy, never easy*. Labels such e, m, h correspond to *easy, medium, hard* classes of AvgC. While labels like l, h correspond to *low, high* classes of StdvC. We show now how to read $\texttt{Scheduler}_5$ represented below. First we order the instances based on 3C classes, from the easiest one to the most difficult one. In each 3C classes we then break ties ordering on the MaxC. The scheduler is executing first the easy (e) then the hard (h) instances of the c1 class. It follows the easy then the hard instances of the c2 class. Finally it schedules the easy then the hard instances of the c3 class.

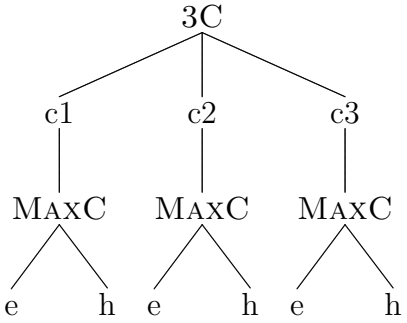


Figure 4.1: $\texttt{Scheduler}_5$ rule

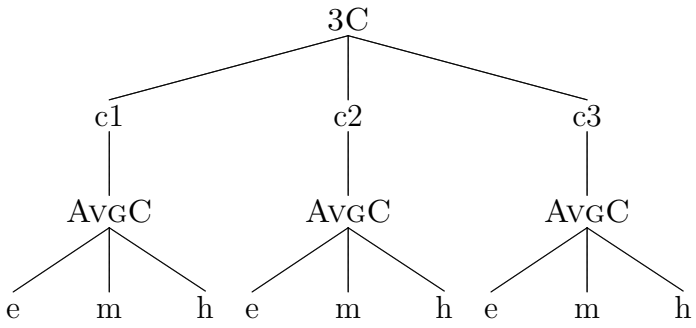


Figure 4.2: $\texttt{Scheduler}_4$ rule

In Figure 4.3 the $\texttt{Scheduler}_1$ rule is illustrated. This rule is using 3C, AvgC and StdvC to order the instances.

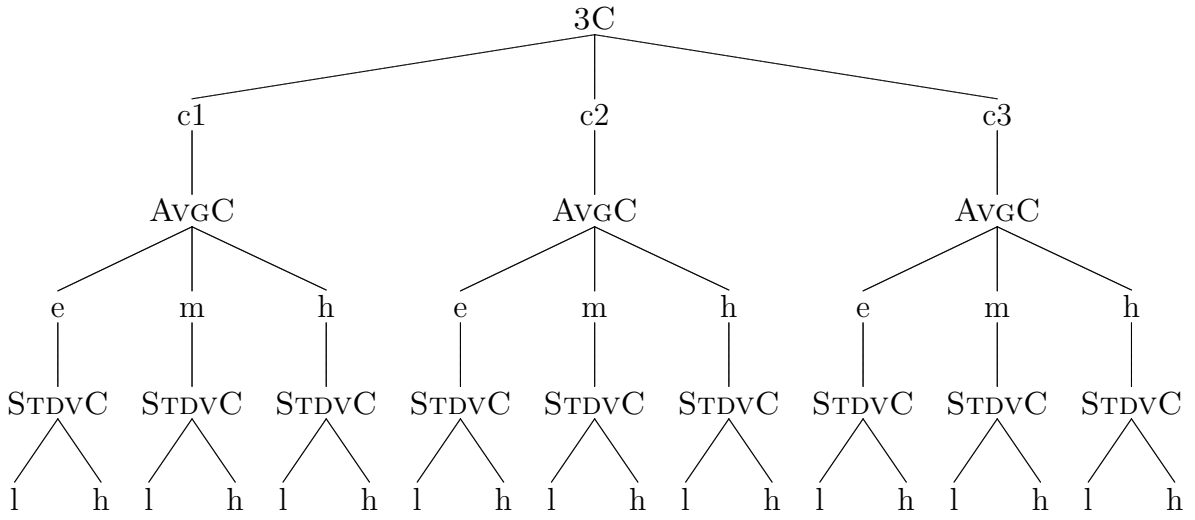


Figure 4.3: `Scheduler`$_1$ rule

`Scheduler`$_3$ is similar to `Scheduler`$_1$, except that MaxC is used instead of StdvC as a tie breaker for 3C. In specific the order is described in the tree in Figure 4.4. The first branch maintains the usual order also helped by the fact that almost all the classes are easy. In the second and third branches a swap of classifiers has been actuated. Indeed, as the numbers will show later, such a swap is improving both the average and median finishing time.

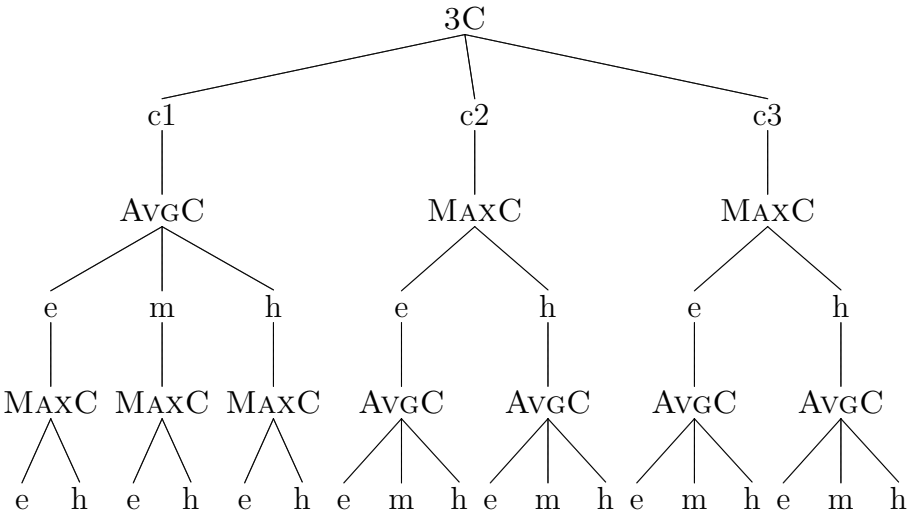


Figure 4.4: `Scheduler`$_3$ rule

Finally `Scheduler`$_2$ employs all the five classifiers as depicted in Figure 4.5. It still applies the same swapping optimization in the `Scheduler`$_3$. For space purpose the ultimate classifiers are restricted: M is MaxC and SD is StdvC .

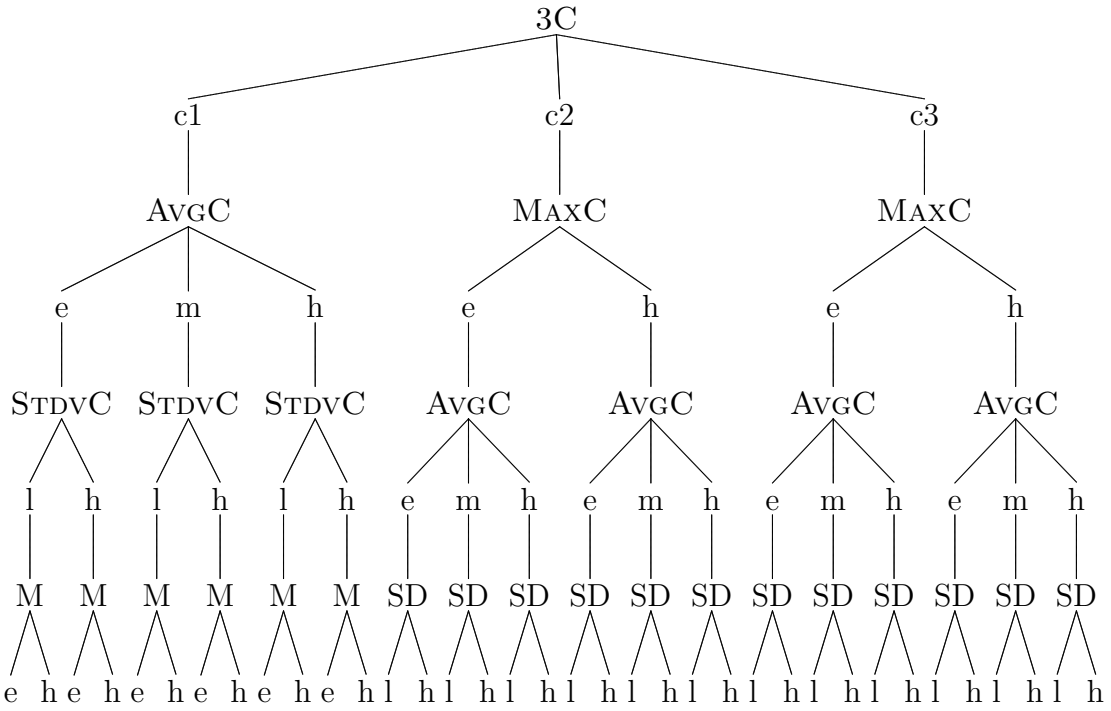


Figure 4.5: `Scheduler`$_2$ rule

We now explain the baseline scheduling rules. The first called `Random`, selects the next instance to solve at random, and solves it with a randomly chosen solver. It is called simply `Random`. The second approach,`Random+FS`, selects the instances randomly and solves each instance with the solver predicted by the Fs classifier. The last baseline approach, `Mistral`, orders the instances based on the MistralC classifier and solves them with the *mistral* solver. The choice of *mistral* has two motivation, first it is the fastest solver of the portfolio and one of the fastest CSP solver in the latest competition. The second reason relies in the simple fact of showing that this portfolio strategy is able to perform better than the single best solver. Surely, this approach has already its weak point in the non termination condition. In fact *mistral* is not able to solve all the instances by itself.

4.1.2 Experimental results

The scheduler is giving a cut off of 1800 seconds. After that an instance is considered timed out. The result of a 250 runs of 5-fold cross validation are summarized in Table 4.1.

Each of the rules just described, except the aforementioned `Mistral` one, assures that each instance is actually solved by the cut off. Saying that, if the solver selected by Fs or randomly times out, the portfolio switches randomly to a new solver until eventually

the termination is reached. This condition is reachable because instances not solvable by the cut off were already excluded from the dataset. Instead, if `Mistral` reaches the cut off, we declare the instance not solved.

Scheduler Category	Scheduler Configuration	Selectors		Finishing Time	
		instance selector	solver selector	average	median
Oracles	Oracle_1	oracle	oracle	922	45
	Oracle_2	3C$\prec$ AvgC$\prec$ StdvC$\prec$ MaxC	oracle	2,637	599
	Oracle_3	3C	oracle	3,468	1,074
Classifiers	Scheduler_2	3C$\prec$ AvgC$\prec$ StdvC$\prec$ MaxC	Fs	6,748	1,389
	Scheduler_3	3C$\prec$ AvgC$\prec$ MaxC	Fs	6,800	1,344
	Scheduler_1	3C$\prec$ AvgC$\prec$ StdvC	Fs	7,051	1,707
	Scheduler_4	3C$\prec$ AvgC	Fs	7,327	2,295
	Scheduler_5	3C$\prec$ MaxC	Fs	7,580	2,247
	Scheduler_6	3C	Fs	8,835	3,731
Baselines	Mistral	MistralC	Mistral time	$\geq$ 15,402	$\geq$ 8,720
	Random+FS	random	Fs	24,344	23,590
	Random	random	random	113,414	113,898

Table 4.1: Performance of classification-based portfolios on a single processor.

The experimental results shows, that, even if the median is slightly higher than the successive $\texttt{Scheduler}_3$ rule, the best average finishing time is reached with the combination of the 3C, AvgC, StdvC, MaxC and Fs classifiers. $\texttt{Scheduler}_2$ clearly gives the top result among the non oracle based scheduling rules. Compared to the `Mistral` rule , we proved that classification based portfolio approach is useful and brings significant measures in runtime. Considering that `Mistral` is not solving every instances in the test, $\texttt{Scheduler}_2$ is increasing the performances in average and medium finishing time respectively of 57% and 84%.

From the observation of $\texttt{Scheduler}_2$ and $\texttt{Oracle}_2$, we can evaluate how effective is Fs classifier compared to the oracle. Certainly Fs is affected by classification errors and this justify the increase of almost 2.5 times in the average finishing time. However, if we consider `Random` and `Random+FS`, we can appreciate how Fs is decreasing the average finishing time of 4.7 times. This states how a random selection is worse than classifier based on selection. We can also observe that the best scheduling rule gives result much closer to the oracle based solutions than the random one. This is testifying that a classification based portfolio is a successful strategy due also to the reliability of the classifiers.

In addition, we have tested how swapping classifiers in two branches of $\texttt{Scheduler}_3$ affects the performances. As Table 4.2 shows, swapping the classifiers yields to decreased average and median finishing time.

Scheduler configuration	Finishing Time	
	average	median
$\texttt{Scheduler}_3$	6,800	1,344
$\texttt{Scheduler}_3$ (no swap)	7,212	2,601

Table 4.2: Performance comparison of $\texttt{Scheduler}_3$ with or without swap.

4.2 Multiple processors case

The second *Java* simulator is actually an extension of the single processor case and it uses the same modus operandi previously introduced:

1. initialization
2. preparation of data structures
3. classification
4. scheduling
5. statistics record
6. structures cleaning

The operations are still the same, from the generic initialization to the preparation of the 5-fold cross validation, creating the training and test sets. Then it still executes the classification of the respective test sets in input and the run of different scheduling rules created, which however will be different. Once it comes to the statistics storage phase new statistical measures are introduced. In fact, moving out from a single to a multiple CPU reality, introduces a series of new possibilities to control, like the makespan or the percentage of unused resources. Those two, indeed, are measures that would not be interesting in a single processor case. The makespan is informally defined as the ending time of the last task. Where obviously in a single processor such a definition would provide the same number over the all runs, in the multi processor case is interesting to analyse all the different ending computation times of each processor. From those computation times we want to extract the maximum, which is the makespan. Such value will show on average how long does it takes to execute the most expensive computation. Knowing the makespan makes interesting to calculate how much of the resources that could be used, are actually left idle.

Given a problem i, the makespan is defined as the maximum of the n processors running time pr.

$$makespan(i) = max(pr_1(i), pr_2(i), \ldots, pr_n(i))$$

Given a problem i, the percentage of unused resources is defined as the difference between the average of ending time and the maximum (the makespan), divided by the makespan.

$$unused_r(i) = \frac{avg(runtime_{pr}(i)) - makespan(i)}{makespan(i)}$$

The larger the difference is, the bigger is the unbalance imbalance in work load among the processors.

Once the statistical values are stored, the simulator cleans the data structures for being able to perform another run of cross validation. Like in the single CPU case, 250 runs are executed and the MLA applied is *Random Committee*. A further explanation of this MLA choice is given later in Figure 4.10.

4.2.1 Scheduling rules

The purpose of this scheduling simulation is to minimise the average finishing time and the scheduling rules already introduced in Section 4.1.1 are still valid in this context. We will however restrict ourselves to only five rules, as shown in Table 4.3: the optimal (`Oracle`$_1$ in the single processor case), the best classification based approach from the single processor (`Scheduler`$_2$) and the baseline rules (`Mistral Random` and `Random+FS`). The job across the machines will be distributed as round robin. In some rules (as shown in Table 4.3), SJF heuristic is employed for dispatching the instances ordered by hardness to the machines. The SJF heuristic is applied with the scheduling rules previously introduced. If the purpose of the scheduling was minimise the makespan, then a technique such as preemption would be the top choice of the scheduling because such method could distribute better the load of computation over the processors involved.

Scheduler configuration			Scheduling rule	
Name	Round Robin	SJF	instance selector	solver selector
Optimal	✓	✓	oracle	oracle
Scheduler$_2$	✓	✓	3C$\prec$ AvgC$\prec$ StdvC$\prec$ MaxC	Fs
Mistral	✓	✓	MistralC	mistral time
Random+FS	✓	✗	random	Fs
Random	✓	✗	random	random

Table 4.3: Rules applied on portfolios in a multiple processors case.

4.2.2 Experimental results

The experimental runs are executed on 5, 10, 25, 50, 75 and 100 processors. Such a number of processors is able to delineate the trend over all the statistics measures.

In Table 4.4 the results over the different scheduling rules are shown. The table reports the results using the average finishing time.

Num. of processors	Optimal	Scheduling2	Mistral	Random+FS	Random
5	201	1299	3096	4719	22711
10	112	664	1590	2382	11461
25	60	285	685	976	4703
50	45	162	386	510	2430
75	41	122	292	351	1684
100	38	102	246	271	1306

Table 4.4: Average finishing time of a scheduling simulation on multiple processors.

The plot in Figure 4.6 shows the trend of the average finishing time. As it is possible to notice, the `Scheduling`$_2$, our top choice, is performing considerably good. The distance between the rule and the `Optimal` is decreasing as the number of processors increases. The same behaviour is notable comparing `Scheduling`$_2$ and `Mistral` where the distance is increasing with the increase of the number of processors. An interesting point is to see that the rule `Random+FS` is asymptotically reaching `Mistral`.

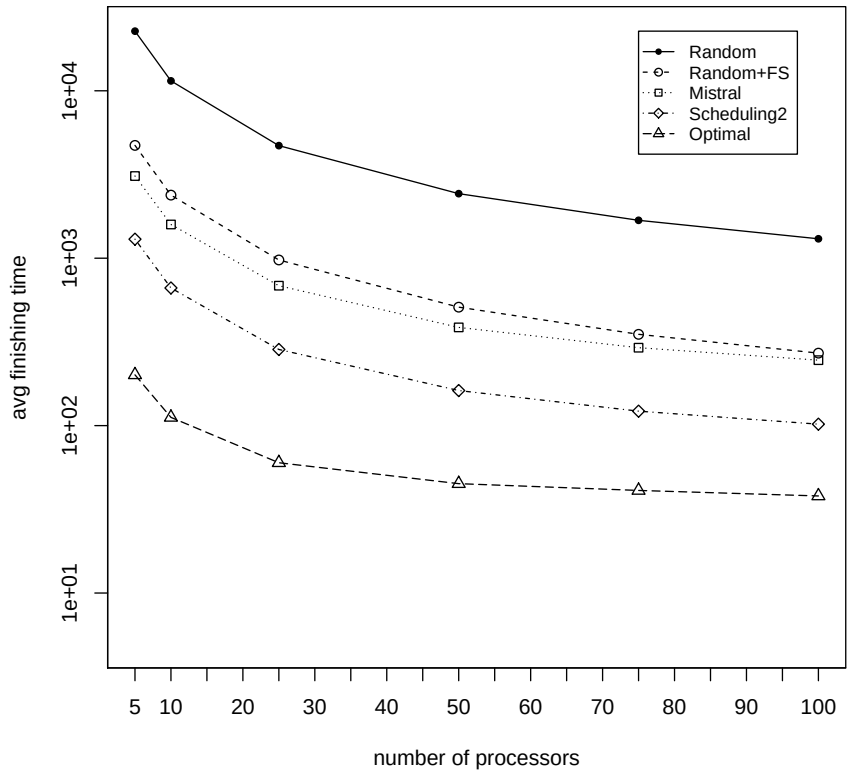


Figure 4.6: Average finishing time.

Before taking a step further to other measures, let us analyse another run of experiments with an exponential scale of processors: 2, 4, 8, 16, 32, 64, 128, 256 and 512. In Figure 4.7, both the x and y axis are log scaled, because we want to show the linearity of the average finishing times. This property says that, as the number of processors increase, the average finishing time is decreasing linearly. This property is maintained until a certain point. In fact, such a linearity is compromised already with 128 processors. In this case the number of processors is getting too high for being able to exploit the power of parallelism in some way and this is mainly due to the number of instances tested.

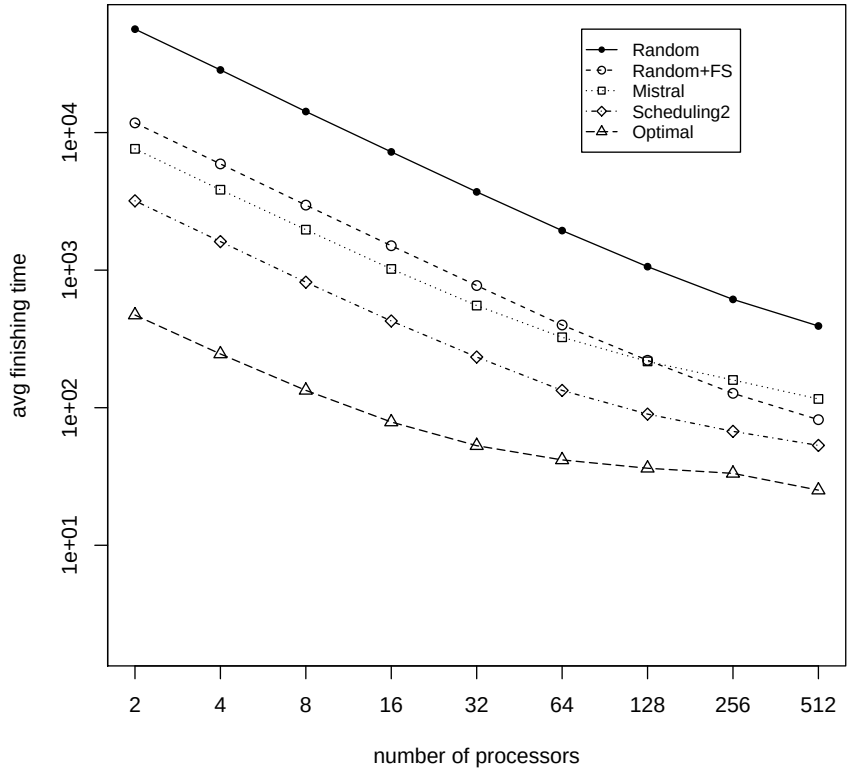


Figure 4.7: Average finishing time on an exponential number of processors.

Num. of processors	Optimal	Scheduling2	Mistral	Random+FS	Random
5	6124	14420	25345	14645	56905
10	3095	8270	13608	8449	31110
25	1256	4194	6220	4308	14531
50	646	2567	3548	2654	8456
75	432	1890	2585	1972	6305
100	321	1515	2073	1590	5140

Table 4.5: Makespan of a scheduling simulation on multiple processors.

In Table (4.5) we report the makespan results.
Comparing the numbers and plotting them in Figure 4.8, we note that ordering the instances randomly or by $\texttt{Scheduler}_2$ rule does not make too much difference from the makespan point of view. What is influencing these similar values is the same selection

strategy of the solver. In the specific the usage of Fs classifier for predicting the solver and the random if the predicted one times-out.

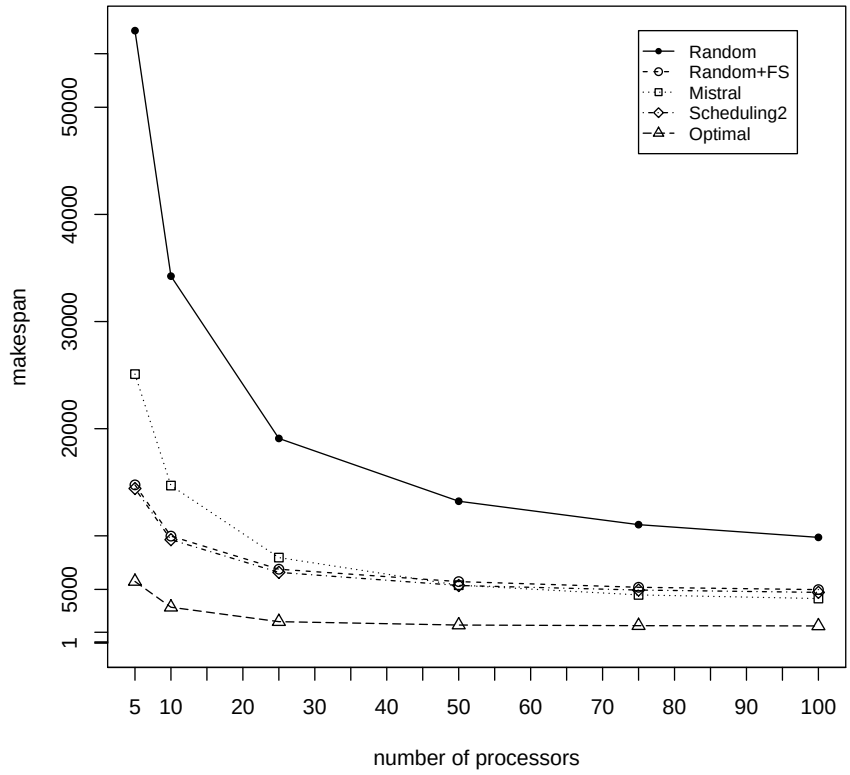


Figure 4.8: Makespan.

In Figure 4.9, we observe the different trends of the unused resources. The increasing value of the `Optimal` over the `Random` and `Mistral` is due to the fact that scheduling with round robin and short job first does not optimize at all the load average on the CPUs. Such a characteristic, like said before, would require different scheduling rules which are not objective of the work done so far. Once again, the similarities between the `Random+FS` and $\texttt{Scheduling}_2$ are due to fact that this measure is based on the makespan.

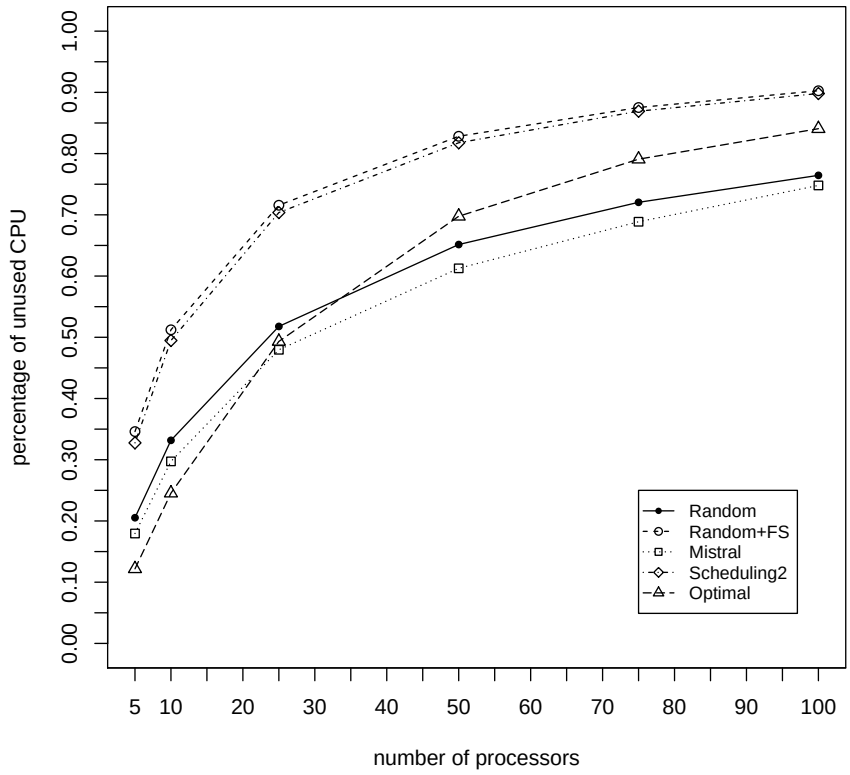


Figure 4.9: Percentage of unused CPU.

As the last experiment, different tests are run with the same scheduling rules ($\texttt{Scheduling}_2$), but this time varying the MLA applied. The purpose is to compare how the MLA are performing on the scheduling, also taking in account the previous classification results. In Section 4.1.2 the classification results showed that *RandomForest* was the algorithm reaching the highest accuracy performances among the MLA set tested. While in this application, algorithms such *RandomCommittee* and *MultiBoost* are actually performing better than *RandomForest*, as the plot in Figure 4.10 shows. The reasons could be multiple, from the difference of cross validation type to the fact that one algorithm once doing misclassification could misclassify worse than another (when it chooses the worst possible error).

Classifier	Toolkit	Implementation
Meta Classifier	WEKA	Random Committee + RandomTree
Decision Tree	WEKA	Random Forest
Meta Classifier	WEKA	MultiBoost + RandomTree
Decision Tree	WEKA	J48

Table 4.6: The classifiers compared on $\texttt{Scheduling}_2$.

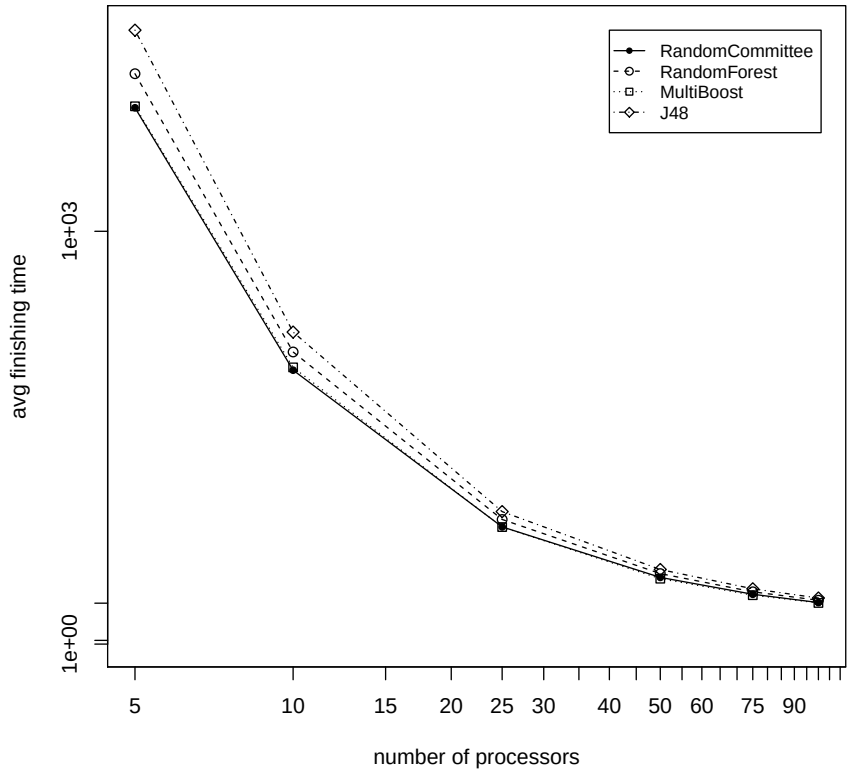


Figure 4.10: Average finishing time compared to different MLA.

4.3 The simulators

The *Java* software developed to simulate the scheduling experiments has the following inheritance structure (Figure 4.11).

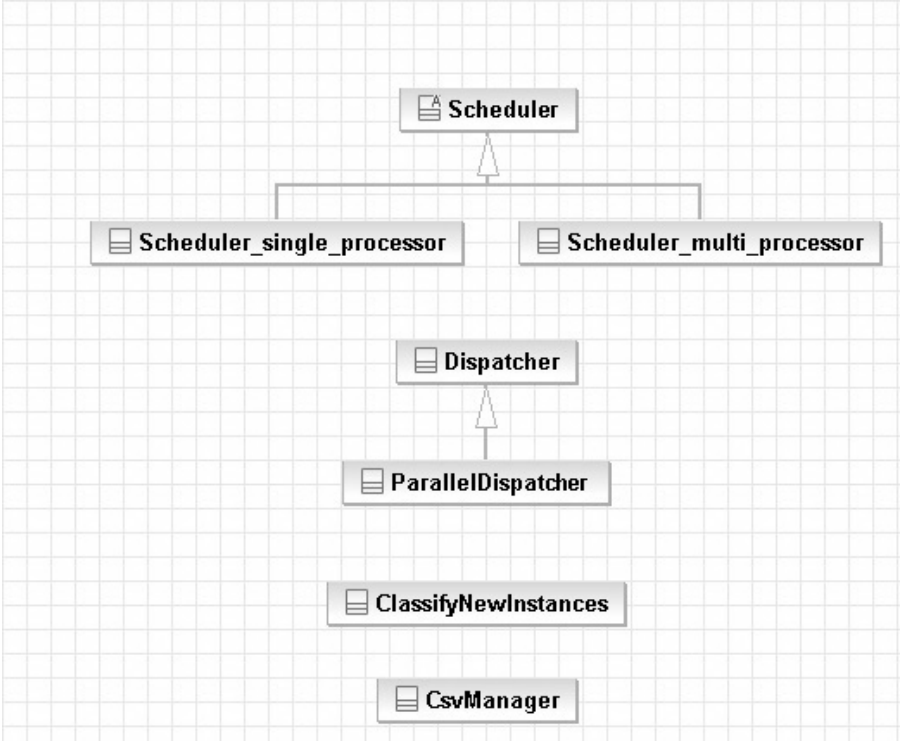


Figure 4.11: UML class diagram depicting inheritance.

In figure 4.11 we introduce the class diagram showing the inheritances among the classes. An empty white arrow indicates an inheritance of the class where the arrow enters, from the class where the arrow leaves. The class `Scheduler` contains abstract methods that have to be implemented by the underlying classes, one for the single processor case and the other for the multi processor case. All the scheduling rules are gathered in the `Dispatcher` class. The same class is extended by `ParallelDispatcher` adding new specialized methods for the scheduling rules in the multiple processor case. In this class we will implement, for example, the round robin strategy. Unrelated to the others, there are the classes `ClassifyNewInstaces` and `CsvManager`. The latter is a simple collection of methods that help to deal with the output CSV (comma separated values) files. While the former contains all the functions able to interact with the Weka tool, from the dataset randomization to the classification operations. Weka[1], in fact, is an open source software written in *Java* and this feature facilitates any integration with an user willing to exploit machine learning algorithms.

In Figure 4.12, the dependencies between the classes are clarified. Each arrow is associated with a label that defines the type of relation between the classes sharing the

[1]http://www.cs.waikato.ac.nz/ml/weka/

same arrow. The relation of dependence has to be read as from the class where the arrow leaves to the class where arrives.

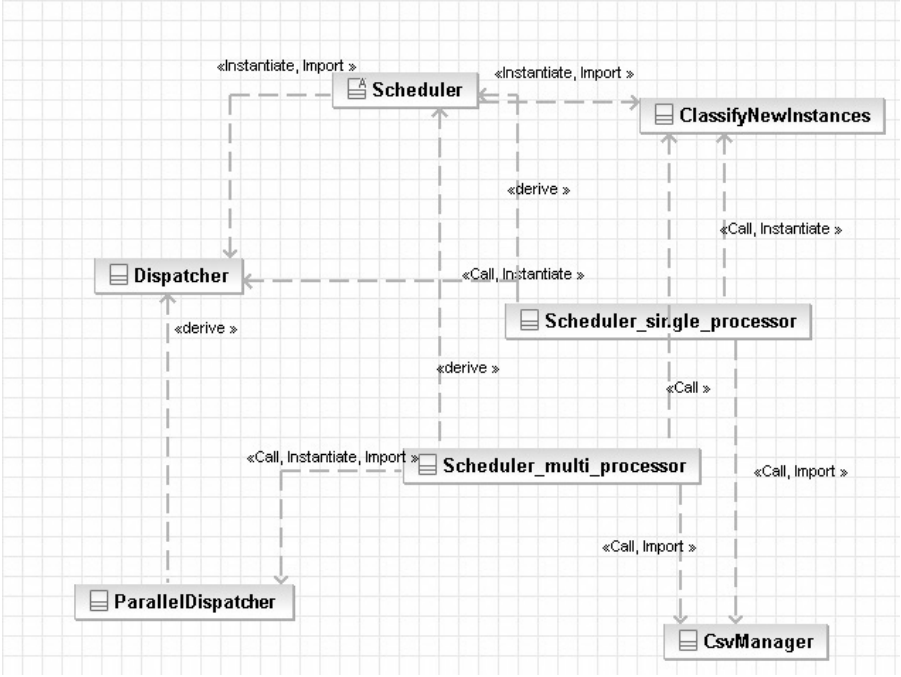


Figure 4.12: UML class diagram depicting dependencies.

The modus operandi of each simulation, described in the previous sections, is here plotted. In each scheduling the main class derived by `Scheduler` (`Scheduler_single_processor` and `Scheduler_multi_processor`) is the main character.

The `Scheduler_single_processor` calls `ClassifyNewInstaces`, `Dispatcher` and `CsvManager` respectively to interact with Weka (operating on the datasets and classifying the new instances), to launch the different scheduling rules and in the end to record the statistical output from the operation just run. `Scheduler_multi_processor` calls the same classes except for the scheduling rules. In this case the extended version, `ParallelDispatcher`, is employed for the scheduling run.

The last thing that needs to be explain are the input and output of the scheduler. In each simulation the input is the dataset expressed in the Weka open file, *arff*. In output both simulations create a *CSV* file for the average finishing time. In the single CPU case an additional file for the median is created, while in the multi CPU case there are two more output for the makespan and the unused resources.

Chapter 5

Related Work

Machine learning, in constraint programming fields, has been used a lot but here we focus especially in application to algorithm tuning and selection.

The hereby presented work utilizes the features of CPHydra for creating a group of classifiers able to give us information on how to use a portfolio of solver to speed up the solving process of CSPs. Combining classifications for solving CSP is a strategy inspired by the good results that classification showed on satisfiability problems. In this paper [7], indeed, it has been shown that, using SATzilla features, a classification technique can be effective on SAT problems.

Our classification based approach differentiates itself from the aforementioned SATzilla, apart from the different nature of the problem (SAT instead of CSP), because SATzilla uses linear regression to forecast the expected running time.

In [38] a portfolio combining multiple heuristics is employed for optimizing techniques to produce a schedule that minimises the total CPU time spent on all instances. The schedule considered is a task-switching schedule, a more powerful class of schedules that allow the proportion of CPU time allocated to each heuristic to change over time. In this paper they tackled a similar idea of computing an offline scheduling and using learning methodology for computing the optimal task-switching scheduling. However the focus of this work is closer to the scheduling optimisation then the learning one.

The adaptive QBF multi engine (AQME) system [32], where a quantified Boolean formulae (QBF) is SAT instances with some quantified variables, uses a similar approach to the portfolio concept: employ different QBF solvers and harness reasoning techniques to learn how to select engine policies. Moreover AQME is a self adaptive QBF solver: a multi engine solver that can update its learned policies in a very similar way to the CBR way. AQME is using classifications based on the features for selecting which en-

gine/solver run. A similar approach is adopted in the work here presented when we use Fs for selecting the portfolio solver.

An example of CBR applied to decision is in [10] where a case based reasoning methodology is used for deciding whether to use constraint programming or integer linear programming as a solving strategy. In the specific, they focus on bid evaluation problem, where it is not easy to determine a priori which is the best technique.

ISAC is an instance specific algorithm configuration that automatically tunes the algorithms by selecting the parameters that yield to best performance [19]. In *ISAC* the solver selection is tackled more specifically. Whether our scheduling rules use Fs classifier for determine which solver to use, *ISAC* is clustering on similarities of the features vector. Then if these clustered features are behaving similarly once they are run with the same solver, they start to tune the algorithm parameters of each cluster. At runtime, when a new instance is submitted in input, it determines the cluster that is closest to the input instance and it solves the instance with the parameters of the respective cluster. In Hydra[42], not to be confused with CPHydra, a framework is built on the combination of automated algorithm configuration and portfolio-based algorithm selection. This complementation of two techniques, using SATzilla as a portfolio builder, is intended for use in problem domains where an adequate set of candidate solvers does not already exist. In [22], given a choice of algorithms and parameter settings, machine learning techniques are used for choosing the algorithm-parameter combination that delivers the best performance for a specific problem. Moreover in this paper a similar procedure to the one described in Chapter 3 has been adopted, but instead of focusing on accuracy and κ, like done here, they focused on the misclassification penalty.

Chapter 6

Conclusion and future work

In the hereby presented work we created a new combination of a portfolio with machine learning techniques. This combination differs from similar work previously introduced, CPHydra and SATzilla. The former is a portfolio for constraint solving that employs a case based reasoning for maximising the probability of solving an instance in a time frame. While the latter is a portfolio for SAT solving that employs a linear regression algorithm to build run time prediction and then choose among its fastest solvers.

Our new portfolio approach is inspired by both portfolios even if more by CPHydra. From it we reused the features, the attributes able to depict a CSP peculiarity. We showed that a classification approach to construct a portfolio of constraint solvers can provide good result. Such a classification would not perform correctly if the features employed for describing the problems would not be efficient. In Chapter 3 we showed that, after optimising the features derived from the portfolio CPHydra, they are statistically significant better at a 95% confidence level compared to the features of an high competitive portfolio solver like SATzilla. In order to show the supremacy of CPHydra features, we first built a group of classifiers based on the run times distribution on the portfolio solvers. Then we run a series of tests using a variety of different machine learning algorithms. The results of these tests are reported in Section 3.6.2. Beyond CPHydra features supremacy, the results shows that the classifiers built are reliable, reaching high values in both accuracy and κ statistics.

Forecasting information by means of classifications, indeed, can improve the execution of portfolio of constraint solvers for solving CSPs. What we showed after is that, the scheduling rule, based on the combination of all classifiers created, is performing well both on single and multiple processor case. In both single and multiple cases we are outperforming a scheduling solution based on the solely fastest solver of the portfolio, *mistral* [16]. Moreover, with the goal of minimising the average finishing time of the instances, the winning strategy is the one that orders the instances on the combination of

four classifiers and uses a fifth one for selecting the solver to run. In a multiple processor scenario, the approach is similar with the only difference that the ordered instances are spread to the machines with a round robin technique.

The hereby presented work resulted to a paper submission to the 2011 International Joint Conference of Artificial Intelligence (IJCAI).[1] By the time of submitting the thesis we are not yet informed about its acceptance status. However the reviewers feedback has revealed that our method is appealing.

A refinement that can be done on the present work regards the multiple processors scenario. It can be interesting to try to change the purpose of minimising the average finishing time to minimise the makespan, and then compare the statistics measures in terms of gain/loose of the two different purposes. Another refinement that could be interesting to analyse is the one related to the solver selection strategy. Right now if a predicted solver goes timeout, another solver randomly chosen from the portfolio, is scheduled on the same processor, until a solution is found. This can be the reason why we obtain such similar makepan values for all those scheduling rules that uses the same strategy. A solution could be schedule the randomly chosen solver to a different processor. Such a strategy would spread the makespan over the solving network.

As a future work there is the extension of the scheduler simulator to a new version able to dynamically execute the problems without knowing them from the beginning. This dynamic behaviour will be first adapted and then included in the service-oriented volunteer computing for massively parallel constraint solving using portfolios presented in [20]. Such an architecture uses a series of different tuning on the portfolio solvers that recalls what has been introduced in some works listed in Chapter 5. The goal of the work in [20] is the one to build an online architecture for solving CSPs. This motives the choice of average finishing time as the performance metric.

[1] http://ijcai-11.iiia.csic.es/

Chapter 7

Acknowledgments

My sincere acknowledgements go to my supervisor professor Zeynep Kiziltan, for giving me the possibility to work on this thesis in Ireland and for the incredible energies spent on my work. I would like to thank professor Barry O'Sullivan and doctor Emmanuel Hebrard for following my project and giving me the possibility to work in their research centre in University College Cork. Thank to all of them and to PhD student Jacopo Mauro for contributing to increase my knowledge and make this thesis work possible. I want to thank Marco Patrignani and Stefania Stefansdottir for their last minute proof reading work and personally for being always present even when the distance is dividing us.

I want to thank all my best friends Nikolas, Dave, Simone, Marco, Laura, Patrizia, Cristina, Andrea, Michele, Lorenzo, Rita and Sonia. Even if I have been abroad most of the time of the past year and half, they were always there for me. Many thanks to all my Erasmus friends with whom I shared fantastic memories and they took me to the position I am currently now. Especially Alessio, Federico, Patrick and Emanuele. I would like to thanks Martina Malafova for being so close and present to me for so long notwithstanding the distance.

My best acknowledgements go to my parents who in primis made all this possible. To my relatives and especially my cousin Cinzia Mandrioli for the continuous support. I already apologize to whomever is missing from the list but the road that brought me here has been a long one. The climb to the mountain is just started.

Bibliography

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59, 1994.

[2] Armando Bazzani, Alessandro Bevilacqua, Dante Bollini, Rosa Brancaccio, Renato Campanini, Nico Lanconelli, Alessandro Riccardi, Davide Romani, and Gianluca Zamboni. Automatic detection of clustered microcalcifications in digital mammograms using an svm classifier. In *ESANN*, pages 195–200, 2000.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[4] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[5] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37, 1960.

[6] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Science/Engineering/Math, 2nd edition, 2003.

[7] David Devlin and Barry O'Sullivan. Satisfiability as a classification problem. In *Proc. of the 19th Irish Conf. on Artificial Intelligence and Cognitive Science*, 2008.

[8] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game-playing agents. In *AAAI*, 2010.

[9] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *ICML*, pages 148–156, 1996.

[10] Cormac Gebruers, Alessio Guerri, Brahim Hnich, and Michela Milano. Making choices using structure at the instance level within a case based reasoning framework. In *CPAIOR*, pages 380–386, 2004.

[11] Cormac Gebruers, Brahim Hnich, Derek G. Bridge, and Eugene C. Freuder. Using cbr to select solution strategies in constraint programming. In *ICCBR*, pages 222–236, 2005.

[12] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.

[13] Eric Guerci, Stefano Ivaldi, Marco Raberto, and Silvano Cincotti. Learning oligopolistic competition in electricity auctions. *Computational Intelligence*, 23(2):197–220, 2007.

[14] Kilem L. Gwet. Inter-rater reliability: Dependency on trait prevalence and marginal homogeneity. Statistical Methods For Inter-Rater Reliability Assessment, October 2002.

[15] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2008.

[16] Emmanuel Hebrard. *Mistral*. http://www.cril.univ-artois.fr/CPAI06/descriptionSolvers/Mistral.pdf, 2006.

[17] G. Holmes, A. Donkin, and I. H. Witten. Weka: a machine learning workbench. pages 357–361, August 1994.

[18] Robert C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63–91, 1993.

[19] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac - instance-specific algorithm configuration. In *ECAI*, pages 751–756, 2010.

[20] Zeynep Kiziltan and Jacopo Mauro. Service-oriented volunteer computing for massively parallel constraint solving using portfolios. In *CPAIOR*, pages 246–251, 2010.

[21] Igor Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in Medicine*, 23(1):89–109, 2001.

[22] Lars Kotthoff, Ian Miguel, and Peter Nightingale. Ensemble classification for constraint solver configuration. In *CP*, pages 321–329, 2010.

[23] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP*, pages 556–572, 2002.

[24] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[25] Donald Michie, David J. Spiegelhalter, and Charles C. Taylor, editors. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, New York, NY, 1994.

[26] Jae H. Min and Youngchan Lee. Bankruptcy prediction using support vector machine with optimal choice of kernel function parameters. *Expert Syst. Appl.*, 28(4):603–614, 2005.

[27] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[28] Stephen Muggleton. Machine learning for systems biology. In *ILP*, pages 416–423, 2005.

[29] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding random sat: Beyond the clauses-to-variables ratio. In *CP*, pages 438–452, 2004.

[30] Eoin OMahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry OSullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. *Proceedings of the 19th Irish Conference on Artificial Intelligence (AICS'08)*, 2009.

[31] Foster Provost. Machine learning from imbalanced data sets 101 (extended abstract).

[32] Luca Pulina and Armando Tacchella. A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints*, 14(1):80–116, 2009.

[33] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[34] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[35] Olivier Roussel and Christophe Lecoutre. Xml representation of constraint networks: Format xcsp 2.1. *CoRR*, abs/0902.2362, 2009.

[36] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

[37] Helmut Simonis. Sudoku as a constraint problem. In Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (CP2005) pp 13-27, October 2005.

[38] Matthew J. Streeter, Daniel Golovin, and Stephen F. Smith. Combining multiple heuristics online. In *AAAI*, pages 1197–1203, 2007.

[39] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[40] Geoffrey I. Webb. Multiboosting: A technique for combining boosting and wagging. *Machine Learning*, 40(2):159–196, 2000.

[41] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, second edition, June 2005.

[42] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *AAAI*, 2010.

[43] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.